18_time-series

June 25, 2022

CPSC 330
Applied Machine Learning

1 Lecture 18: Time series

UBC 2022 Summer

Instructor: Mehrdad Oveisi

1.1 Imports

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.compose import ColumnTransformer, make_column_transformer
from sklearn.dummy import DummyClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import (
    TimeSeriesSplit,
    cross_val_score,
    cross_validate,
    train_test_split,
)
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, StandardScaler

plt.rcParams["font.size"] = 16
from datetime import datetime
```

1.2 Learning objectives

- Explain the pitfalls of train/test splitting with time series data.
- Appropriately split time series data, both train/test split and cross-validation.

- Perform time series feature engineering:
  - Encode time as various features in a tabular dataset
  - Create lag-based features
- Explain how can you forecast multiple time steps into the future.
- Explain the challenges of time series data with unequally spaced time points.
- At a high level, explain the concepts of seasonality and trends.

1.3 Motivation

- **Time series** is a collection of data points indexed in time order.
- Time series is everywhere:
  - Physical sciences (e.g., weather forecasting)

  - Economics, finance (e.g., stocks, market trends)
  - Engineering (e.g., energy consumption)
  - Social sciences
  - Sports analytics

Let's start with a simple example from Introduction to Machine Learning with Python book.

In New York city there is a network of bike rental stations with a subscription system. The stations are all around the city. The anonymized data is available here.

The task we will focus on is predicting how many people will rent a bicycle from a particular station for a given time and day. We might be interested in knowing this so that we know whether there will be any bikes left at the station for a particular day and time.

[2]:
```
import mglearn

citibike = mglearn.datasets.load_citibike()
citibike.head()
```

[2]:
```
starttime
2015-08-01 00:00:00     3
2015-08-01 03:00:00     0
2015-08-01 06:00:00     9
2015-08-01 09:00:00    41
2015-08-01 12:00:00    39
Freq: 3H, Name: one, dtype: int64
```

- The only feature we have is the date time feature.
  - Example: 2015-08-01 00:00:00
- The target is the number of rentals in the next 3 hours.
  - Example: 3 rentals between 2015-08-01 00:00:00 and 2015-08-01 03:00:00

[3]:
```
citibike.index.min()
```

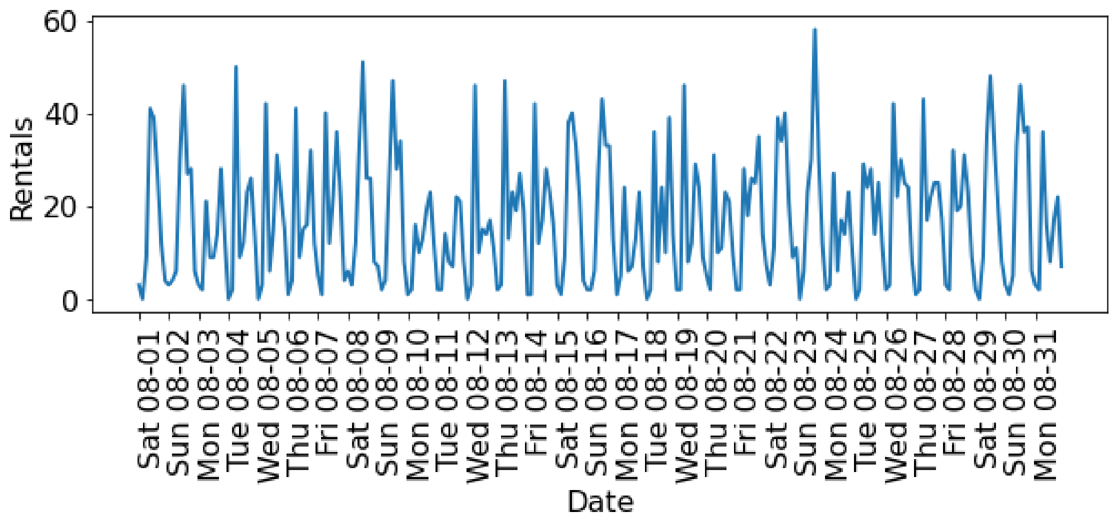
[3]:
```
Timestamp('2015-08-01 00:00:00', freq='3H')
```

[4]:
```
citibike.index.max()
```

[4]: Timestamp('2015-08-31 21:00:00', freq='3H')

We have data for August 2015.

[5]:
```
plt.figure(figsize=(10, 3))
xticks = pd.date_range(start=citibike.index.min(), end=citibike.index.max(),␣
 ↪freq="D")
plt.xticks(xticks, xticks.strftime("%a %m-%d"), rotation=90, ha="left")
plt.plot(citibike, linewidth=2)
plt.xlabel("Date")
plt.ylabel("Rentals");
```



- We see the day and night pattern
- We see the weekend and weekday pattern

- Questions you might want to answer: How many people are likely to rent a bike at this station tomorrow at 3pm given everything we know about rentals in the past?
- We want to learn from the past and predict the future.

1.3.1 Train/test split for temporal data

- What will happen if we split this data the usual way?

[6]:
```
train_df, test_df = train_test_split(citibike, test_size=0.2, random_state=123)
```

[7]:
```
test_df.head()
```

[7]:
```
starttime
2015-08-26 12:00:00    30
2015-08-12 09:00:00    10
2015-08-19 03:00:00     2
```

```
2015-08-07 12:00:00    22
2015-08-03 09:00:00     9
Name: one, dtype: int64
```

[8]:
```
train_df.index.max()
```

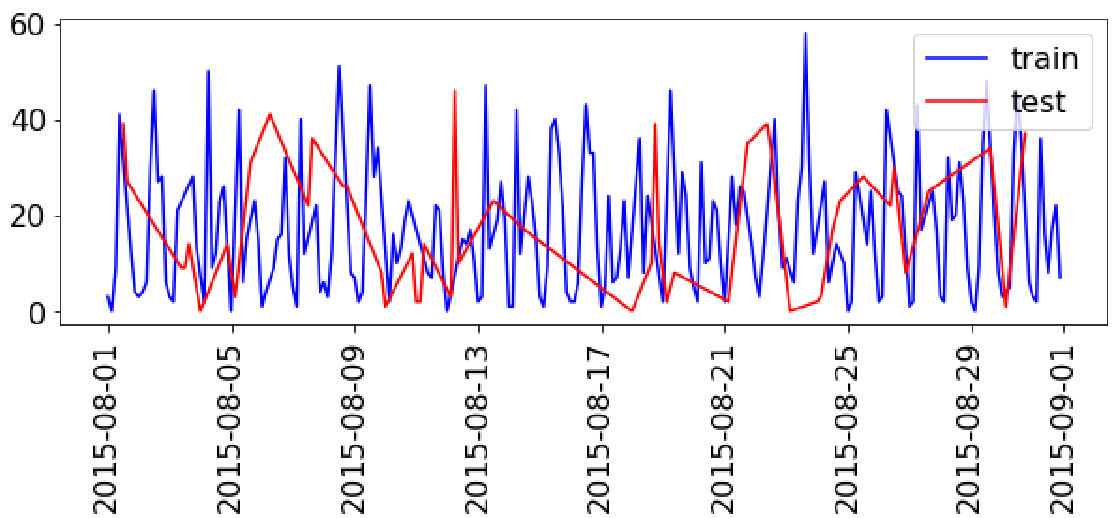
[8]:
```
Timestamp('2015-08-31 21:00:00')
```

[9]:
```
test_df.index.min()
```

[9]:
```
Timestamp('2015-08-01 12:00:00')
```

- So, we are training on data that came after our test data!
- If we want to forecast, **we aren't allowed to know what happened in the future**!
- There may be cases where this is OK, e.g. if you aren't trying to forecast and just want to understand your data (maybe you're not even splitting).
- But, for our purposes, we want to avoid this.

[10]:
```
plt.figure(figsize=(10, 3))
train_df_sort = train_df.sort_index()
test_df_sort = test_df.sort_index()

plt.plot(train_df_sort, "b", label="train")
plt.plot(test_df_sort, "r", label="test")
plt.xticks(rotation="vertical")
plt.legend();
```



We'll split the data as follows:

- We have total 248 data points.

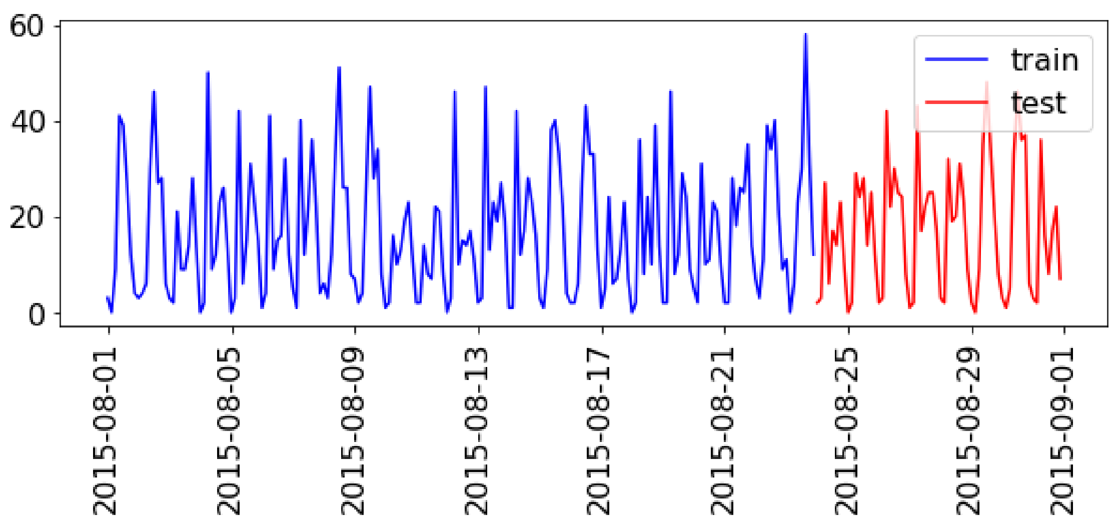
- We'll use the fist 184 data points corresponding to the first 23 days as training data - And the remaining 64 data points corresponding to the remaining 8 days as test data.

[11]:
```
citibike.shape
```

[11]: (248,)

[12]:
```
n_train = 184
train_df = citibike[:184]
test_df = citibike[184:]
```

[13]:
```
plt.figure(figsize=(10, 3))
train_df_sort = train_df.sort_index()
test_df_sort = test_df.sort_index()

plt.plot(train_df_sort, "b", label="train")
plt.plot(test_df_sort, "r", label="test")
plt.xticks(rotation="vertical")
plt.legend();
```



- This split is looking reasonable now.

1.3.2 Training models

- In this toy data, we just have a single feature: the date time feature.
- We need to encode this feature if we want to build machine learning models.
- A common way that dates are stored on computers is using POSIX time, which is the number of seconds since January 1970 00:00:00 (this is beginning of Unix time).
- Let's start with encoding this feature as a single integer representing this POSIX time.

[14]:
```python
X = (
    citibike.index.astype("int64").values.reshape(-1, 1) // 10 ** 9
)  # convert to POSIX time by dividing by 10**9
y = citibike.values
```

[15]:
```python
y_train = train_df.values
y_test = test_df.values
# convert to POSIX time by dividing by 10**9
X_train = train_df.index.astype("int64").values.reshape(-1, 1) // 10 ** 9
X_test = test_df.index.astype("int64").values.reshape(-1, 1) // 10 ** 9
```

[16]:
```python
X_train[:10]
```

[16]:
```
array([[1438387200],
       [1438398000],
       [1438408800],
       [1438419600],
       [1438430400],
       [1438441200],
       [1438452000],
       [1438462800],
       [1438473600],
       [1438484400]])
```

[17]:
```python
y_train[:10]
```

[17]:
```
array([ 3,  0,  9, 41, 39, 27, 12,  4,  3,  4])
```

- Our prediction task is a regression task.

Let's try random forest regression.

[18]:
```python
from sklearn.ensemble import RandomForestRegressor

regressor = RandomForestRegressor(n_estimators=100, random_state=0)
regressor.fit(X_train, y_train)

print("Train-set R^2: {:.2f}".format(regressor.score(X_train, y_train)))
print("Test-set R^2: {:.2f}".format(regressor.score(X_test, y_test)))
```

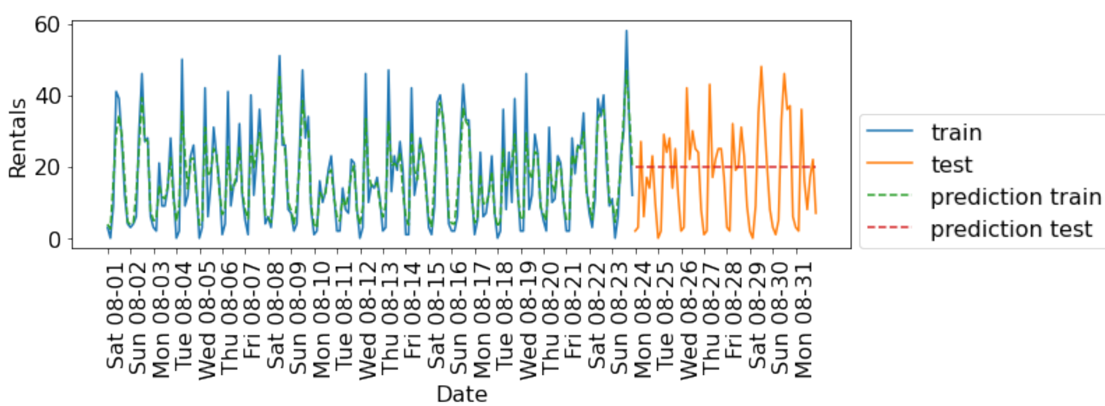
```
Train-set R^2: 0.85
Test-set R^2: -0.04
```

[19]:
```python
## Code credit: https://learning.oreilly.com/library/view/
 ↪introduction-to-machine/9781449369880/


def eval_on_features(features, target, regressor):
    # split the given features into a training and a test set
```

```
    X_train, X_test = features[:n_train], features[n_train:]
    # also split the target array
    y_train, y_test = target[:n_train], target[n_train:]
    regressor.fit(X_train, y_train)
    print("Train-set R^2: {:.2f}".format(regressor.score(X_train, y_train)))
    print("Test-set R^2: {:.2f}".format(regressor.score(X_test, y_test)))
    y_pred = regressor.predict(X_test)
    y_pred_train = regressor.predict(X_train)
    plt.figure(figsize=(10, 3))

    plt.xticks(range(0, len(X), 8), xticks.strftime("%a %m-%d"), rotation=90,␣
 ↪ha="left")

    plt.plot(range(n_train), y_train, label="train")
    plt.plot(range(n_train, len(y_test) + n_train), y_test, "-", label="test")
    plt.plot(range(n_train), y_pred_train, "--", label="prediction train")

    plt.plot(
        range(n_train, len(y_test) + n_train), y_pred, "--", label="prediction␣
 ↪test"
    )
    plt.legend(loc=(1.01, 0))
    plt.xlabel("Date")
    plt.ylabel("Rentals")
```

[20]:
```
from sklearn.ensemble import RandomForestRegressor

regressor = RandomForestRegressor(n_estimators=100, random_state=0)
eval_on_features(X, y, regressor)
```

```
Train-set R^2: 0.85
Test-set R^2: -0.04
```



- The predictions on the training score is pretty good
- But for the test data, a constant line is predicted …
- What's going on?

- The model is based on only one feature: POSIX time feature.
- And the value of the POSIX time feature is outside the range of the feature values in the training set.
- Tree-based models cannot *extrapolate* to feature ranges outside the training data.
- The model predicted the target value of the closest point in the training set.

Can we come up with better features?

1.3.3 Feature engineering for date/time columns

- Note that our index is of this special type: `DateTimeIndex`. We can extract all kinds of interesting information from it.

[21]:
```
citibike.index
```

[21]:
```
DatetimeIndex(['2015-08-01 00:00:00', '2015-08-01 03:00:00',
               '2015-08-01 06:00:00', '2015-08-01 09:00:00',
               '2015-08-01 12:00:00', '2015-08-01 15:00:00',
               '2015-08-01 18:00:00', '2015-08-01 21:00:00',
               '2015-08-02 00:00:00', '2015-08-02 03:00:00',
               …
               '2015-08-30 18:00:00', '2015-08-30 21:00:00',
               '2015-08-31 00:00:00', '2015-08-31 03:00:00',
               '2015-08-31 06:00:00', '2015-08-31 09:00:00',
               '2015-08-31 12:00:00', '2015-08-31 15:00:00',
               '2015-08-31 18:00:00', '2015-08-31 21:00:00'],
              dtype='datetime64[ns]', name='starttime', length=248, freq='3H')
```

[22]:
```
citibike.index.month_name()
```

[22]:
```
Index(['August', 'August', 'August', 'August', 'August', 'August', 'August',
       'August', 'August', 'August',
       …
       'August', 'August', 'August', 'August', 'August', 'August', 'August',
       'August', 'August', 'August'],
      dtype='object', name='starttime', length=248)
```

[23]:
```
citibike.index.dayofweek
```

[23]:
```
Int64Index([5, 5, 5, 5, 5, 5, 5, 5, 6, 6,
            …
            6, 6, 0, 0, 0, 0, 0, 0, 0, 0],
           dtype='int64', name='starttime', length=248)
```

[24]:
```
citibike.index.day_name()
```

```
[24]: Index(['Saturday', 'Saturday', 'Saturday', 'Saturday', 'Saturday', 'Saturday',
              'Saturday', 'Saturday', 'Sunday', 'Sunday',
              …
              'Sunday', 'Sunday', 'Monday', 'Monday', 'Monday', 'Monday', 'Monday',
              'Monday', 'Monday', 'Monday'],
             dtype='object', name='starttime', length=248)
```

```
[25]: citibike.index.hour
```

```
[25]: Int64Index([ 0,  3,  6,  9, 12, 15, 18, 21,  0,  3,
                   …
                   18, 21,  0,  3,  6,  9, 12, 15, 18, 21],
                  dtype='int64', name='starttime', length=248)
```

- We noted before that the time of the day and day of the week seem quite important.
- Let's add these two features.

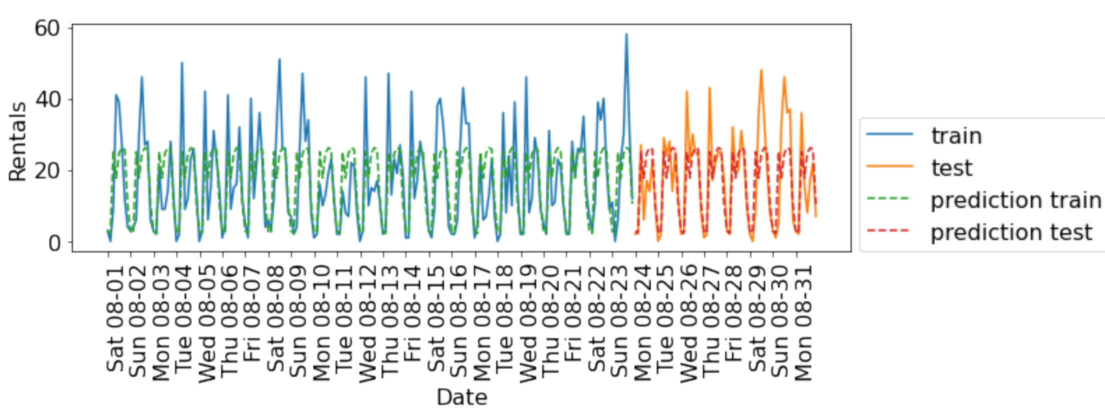
Let's first add the time of the day.

```
[26]: X_hour = citibike.index.hour.values.reshape(-1, 1)
      X_hour[:10]
```

```
[26]: array([[ 0],
             [ 3],
             [ 6],
             [ 9],
             [12],
             [15],
             [18],
             [21],
             [ 0],
             [ 3]])
```

```
[27]: eval_on_features(X_hour, y, regressor)
```

```
Train-set R^2: 0.50
Test-set R^2: 0.60
```



The scores are better than before.

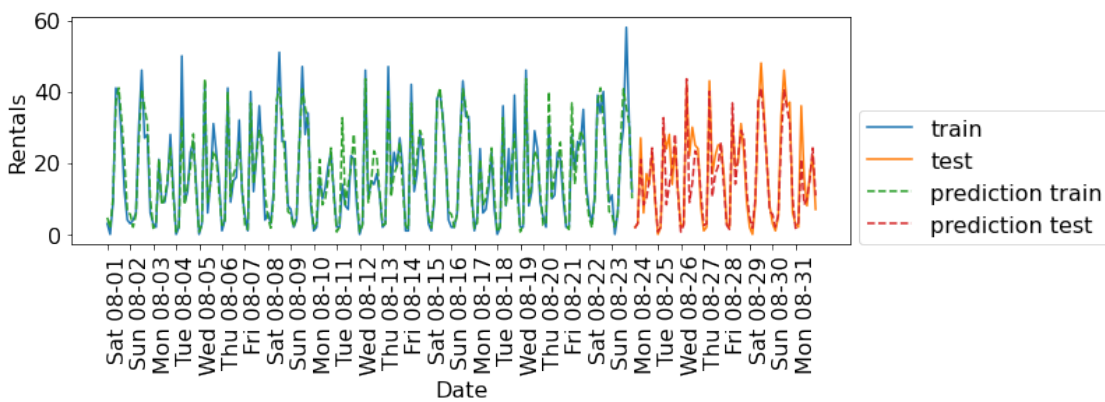
Now let's add day of the week along with time of the day.

[28]:
```
X_hour_week = np.hstack([
    citibike.index.dayofweek.values.reshape(-1, 1),
    citibike.index.hour.values.reshape(-1, 1),])

X_hour_week[:5]
```

[28]:
```
array([[ 5,  0],
       [ 5,  3],
       [ 5,  6],
       [ 5,  9],
       [ 5, 12]])
```

[29]:
```
eval_on_features(X_hour_week, y, regressor)
```

```
Train-set R^2: 0.89
Test-set R^2: 0.84
```

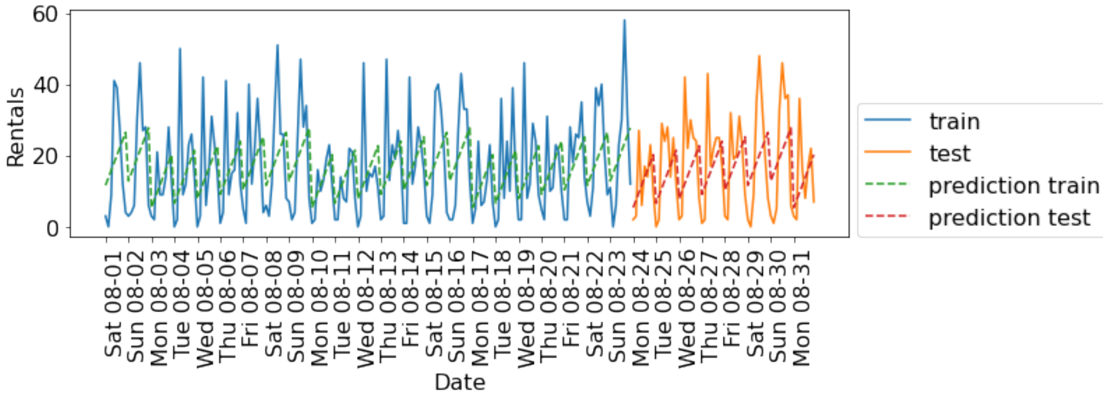


The results are much better. The time of the day and day of the week features are clearly helping.

- Do we need a complex model such as a random forest?
- Let's try `Ridge` with these features.

[30]:
```
from sklearn.linear_model import Ridge

lr = Ridge()
eval_on_features(X_hour_week, y, lr)
```

```
Train-set R^2: 0.16
Test-set R^2: 0.13
```



- Ridge is performing **poorly** on the training as well as test data.
- It's not able to capture the periodic pattern.
- The reason is that we have encoded **time of day using integers**.
- A linear function can only learn a linear function of the time of day.
- What if we encode this feature as a **categorical** variable?

[31]:
```
enc = OneHotEncoder()
X_hour_week_onehot = enc.fit_transform(X_hour_week).toarray()
X_hour_week_onehot.shape, X_hour_week.shape
```

[31]:
```
((248, 15), (248, 2))
```

[32]:
```
enc.get_feature_names_out(['day','hour'])
```

[32]:
```
array(['day_0', 'day_1', 'day_2', 'day_3', 'day_4', 'day_5', 'day_6',
       'hour_0', 'hour_3', 'hour_6', 'hour_9', 'hour_12', 'hour_15',
       'hour_18', 'hour_21'], dtype=object)
```

Or in a more readable format:

[33]:
```
hour = ["%02d:00" % i for i in range(0, 24, 3)]
day = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
features_onehot = day + hour
print(features_onehot)
```

```
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun', '00:00', '03:00', '06:00',
'09:00', '12:00', '15:00', '18:00', '21:00']
```

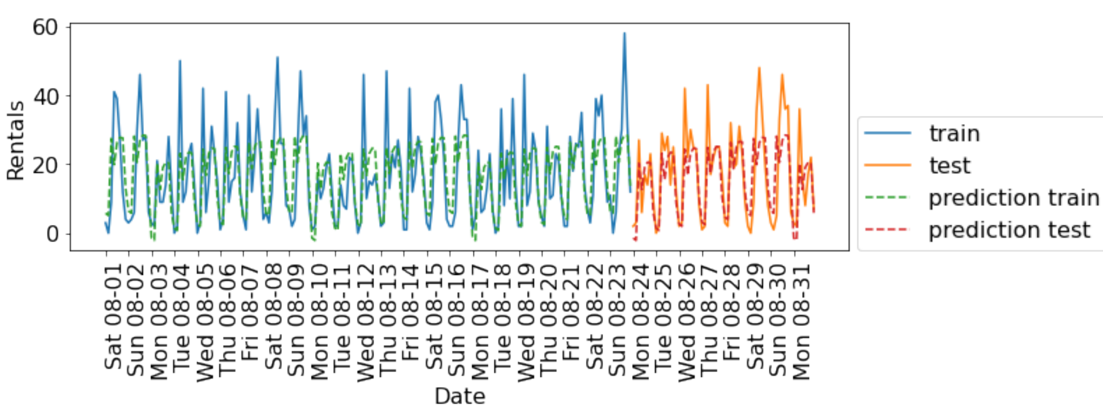
[34]:
```
pd.DataFrame(X_hour_week_onehot, columns=features_onehot)
```

[34]:
```
    Mon  Tue  Wed  Thu  Fri  Sat  Sun  00:00  03:00  06:00  09:00  12:00  \
0   0.0  0.0  0.0  0.0  0.0  1.0  0.0    1.0    0.0    0.0    0.0    0.0
1   0.0  0.0  0.0  0.0  0.0  1.0  0.0    0.0    1.0    0.0    0.0    0.0
2   0.0  0.0  0.0  0.0  0.0  1.0  0.0    0.0    0.0    1.0    0.0    0.0
```

```
3    0.0  0.0  0.0  0.0  0.0  1.0  0.0    0.0    0.0    0.0    1.0    0.0
4    0.0  0.0  0.0  0.0  0.0  1.0  0.0    0.0    0.0    0.0    0.0    1.0
..   …   …   …   …   …   …   …     …     …     …     …     …
243  1.0  0.0  0.0  0.0  0.0  0.0  0.0    0.0    0.0    0.0    1.0    0.0
244  1.0  0.0  0.0  0.0  0.0  0.0  0.0    0.0    0.0    0.0    0.0    1.0
245  1.0  0.0  0.0  0.0  0.0  0.0  0.0    0.0    0.0    0.0    0.0    0.0
246  1.0  0.0  0.0  0.0  0.0  0.0  0.0    0.0    0.0    0.0    0.0    0.0
247  1.0  0.0  0.0  0.0  0.0  0.0  0.0    0.0    0.0    0.0    0.0    0.0

     15:00  18:00  21:00
0      0.0    0.0    0.0
1      0.0    0.0    0.0
2      0.0    0.0    0.0
3      0.0    0.0    0.0
4      0.0    0.0    0.0
..     …     …     …
243    0.0    0.0    0.0
244    0.0    0.0    0.0
245    1.0    0.0    0.0
246    0.0    1.0    0.0
247    0.0    0.0    1.0

[248 rows x 15 columns]
```

[35]:
```
eval_on_features(X_hour_week_onehot, y, Ridge())
```

```
Train-set R^2: 0.53
Test-set R^2: 0.62
```



- What if we add **interaction features**?
- We can do it using `sklearn`'s `PolynomialFeatures` transformer.

[36]:
```python
from sklearn.preprocessing import PolynomialFeatures

poly_transformer = PolynomialFeatures(
    degree=2, interaction_only=True, include_bias=False
)
```

[37]:
```python
toy = np.arange(12).reshape(-1, 3, order='F')
pd.DataFrame(toy, columns=['A','B','C',])
```

[37]:

	A	B	C
0	0	4	8
1	1	5	9
2	2	6	10
3	3	7	11

[38]:
```python
pd.DataFrame(poly_transformer.fit_transform(toy),
             columns=poly_transformer.get_feature_names_out(['A','B','C']))
```

[38]:

	A	B	C	A B	A C	B C
0	0.0	4.0	8.0	0.0	0.0	32.0
1	1.0	5.0	9.0	5.0	9.0	45.0
2	2.0	6.0	10.0	12.0	20.0	60.0
3	3.0	7.0	11.0	21.0	33.0	77.0

[39]:
```python
X_hour_week_onehot_poly = poly_transformer.fit_transform(X_hour_week_onehot)
pd.DataFrame(X_hour_week_onehot_poly).head()
```

[39]:

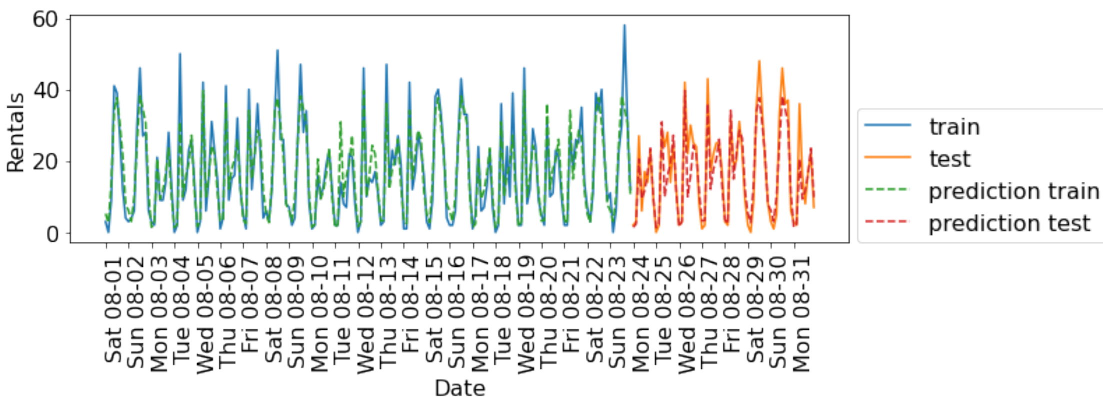
	0	1	2	3	4	5	6	7	8	9	…	110	111	112	113	\
0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	…	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	…	0.0	0.0	0.0	0.0	
2	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	…	0.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	…	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	…	0.0	0.0	0.0	0.0	

	114	115	116	117	118	119
0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0

[5 rows x 120 columns]

[40]:
```python
lr = Ridge()
eval_on_features(X_hour_week_onehot_poly, y, lr)
```

```
Train-set R^2: 0.87
Test-set R^2: 0.85
```



[41]:
```
X_hour_week_onehot_poly.shape, X_hour_week_onehot.shape, X_hour_week.shape
```

[41]: ((248, 120), (248, 15), (248, 2))

Let's see what are the column names for `X_hour_week_onehot_poly`

[42]:
```
print(features_onehot)
```

```
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun', '00:00', '03:00', '06:00',
'09:00', '12:00', '15:00', '18:00', '21:00']
```

[43]:
```
features_onehot_poly = poly_transformer.get_feature_names_out(features_onehot)
features_onehot_poly
```

[43]:
```
array(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun', '00:00', '03:00',
       '06:00', '09:00', '12:00', '15:00', '18:00', '21:00', 'Mon Tue',
       'Mon Wed', 'Mon Thu', 'Mon Fri', 'Mon Sat', 'Mon Sun', 'Mon 00:00',
       'Mon 03:00', 'Mon 06:00', 'Mon 09:00', 'Mon 12:00', 'Mon 15:00',
       'Mon 18:00', 'Mon 21:00', 'Tue Wed', 'Tue Thu', 'Tue Fri',
       'Tue Sat', 'Tue Sun', 'Tue 00:00', 'Tue 03:00', 'Tue 06:00',
       'Tue 09:00', 'Tue 12:00', 'Tue 15:00', 'Tue 18:00', 'Tue 21:00',
       'Wed Thu', 'Wed Fri', 'Wed Sat', 'Wed Sun', 'Wed 00:00',
       'Wed 03:00', 'Wed 06:00', 'Wed 09:00', 'Wed 12:00', 'Wed 15:00',
       'Wed 18:00', 'Wed 21:00', 'Thu Fri', 'Thu Sat', 'Thu Sun',
       'Thu 00:00', 'Thu 03:00', 'Thu 06:00', 'Thu 09:00', 'Thu 12:00',
       'Thu 15:00', 'Thu 18:00', 'Thu 21:00', 'Fri Sat', 'Fri Sun',
       'Fri 00:00', 'Fri 03:00', 'Fri 06:00', 'Fri 09:00', 'Fri 12:00',
       'Fri 15:00', 'Fri 18:00', 'Fri 21:00', 'Sat Sun', 'Sat 00:00',
       'Sat 03:00', 'Sat 06:00', 'Sat 09:00', 'Sat 12:00', 'Sat 15:00',
       'Sat 18:00', 'Sat 21:00', 'Sun 00:00', 'Sun 03:00', 'Sun 06:00',
       'Sun 09:00', 'Sun 12:00', 'Sun 15:00', 'Sun 18:00', 'Sun 21:00',
       '00:00 03:00', '00:00 06:00', '00:00 09:00', '00:00 12:00',
       '00:00 15:00', '00:00 18:00', '00:00 21:00', '03:00 06:00',
       '03:00 09:00', '03:00 12:00', '03:00 15:00', '03:00 18:00',
```

```
       '03:00 21:00', '06:00 09:00', '06:00 12:00', '06:00 15:00',
       '06:00 18:00', '06:00 21:00', '09:00 12:00', '09:00 15:00',
       '09:00 18:00', '09:00 21:00', '12:00 15:00', '12:00 18:00',
       '12:00 21:00', '15:00 18:00', '15:00 21:00', '18:00 21:00'],
      dtype=object)
```

Let's examine the coefficients learned by `Ridge`. Note that many of the coefficients are zeros.

```
[44]: np.sum(lr.coef_ == 0), len(lr.coef_)
```

```
[44]: (49, 120)
```

```
[45]: np.array(features_onehot_poly)[lr.coef_ == 0]
```

```
[45]: array(['Mon Tue', 'Mon Wed', 'Mon Thu', 'Mon Fri', 'Mon Sat', 'Mon Sun',
       'Tue Wed', 'Tue Thu', 'Tue Fri', 'Tue Sat', 'Tue Sun', 'Wed Thu',
       'Wed Fri', 'Wed Sat', 'Wed Sun', 'Thu Fri', 'Thu Sat', 'Thu Sun',
       'Fri Sat', 'Fri Sun', 'Sat Sun', '00:00 03:00', '00:00 06:00',
       '00:00 09:00', '00:00 12:00', '00:00 15:00', '00:00 18:00',
       '00:00 21:00', '03:00 06:00', '03:00 09:00', '03:00 12:00',
       '03:00 15:00', '03:00 18:00', '03:00 21:00', '06:00 09:00',
       '06:00 12:00', '06:00 15:00', '06:00 18:00', '06:00 21:00',
       '09:00 12:00', '09:00 15:00', '09:00 18:00', '09:00 21:00',
       '12:00 15:00', '12:00 18:00', '12:00 21:00', '15:00 18:00',
       '15:00 21:00', '18:00 21:00'], dtype=object)
```

```
[46]: np.array(features_onehot_poly)[lr.coef_ != 0]
```

```
[46]: array(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun', '00:00', '03:00',
       '06:00', '09:00', '12:00', '15:00', '18:00', '21:00', 'Mon 00:00',
       'Mon 03:00', 'Mon 06:00', 'Mon 09:00', 'Mon 12:00', 'Mon 15:00',
       'Mon 18:00', 'Mon 21:00', 'Tue 00:00', 'Tue 03:00', 'Tue 06:00',
       'Tue 09:00', 'Tue 12:00', 'Tue 15:00', 'Tue 18:00', 'Tue 21:00',
       'Wed 00:00', 'Wed 03:00', 'Wed 06:00', 'Wed 09:00', 'Wed 12:00',
       'Wed 15:00', 'Wed 18:00', 'Wed 21:00', 'Thu 00:00', 'Thu 03:00',
       'Thu 06:00', 'Thu 09:00', 'Thu 12:00', 'Thu 15:00', 'Thu 18:00',
       'Thu 21:00', 'Fri 00:00', 'Fri 03:00', 'Fri 06:00', 'Fri 09:00',
       'Fri 12:00', 'Fri 15:00', 'Fri 18:00', 'Fri 21:00', 'Sat 00:00',
       'Sat 03:00', 'Sat 06:00', 'Sat 09:00', 'Sat 12:00', 'Sat 15:00',
       'Sat 18:00', 'Sat 21:00', 'Sun 00:00', 'Sun 03:00', 'Sun 06:00',
       'Sun 09:00', 'Sun 12:00', 'Sun 15:00', 'Sun 18:00', 'Sun 21:00'],
      dtype=object)
```

```
[47]: features_nonzero = np.array(features_onehot_poly)[lr.coef_ != 0]
      coef_nonzero = lr.coef_[lr.coef_ != 0]
```

```
[48]: pd.DataFrame(coef_nonzero, index=features_nonzero, columns=["Coefficient"]).
       ↪sort_values(
```

```
    "Coefficient", ascending=False
)
```

[48]:

```
             Coefficient
Sat 09:00      15.196739
Wed 06:00      15.005809
Sat 12:00      13.437684
Sun 12:00      13.362009
Thu 06:00      10.907595
…                    …
Sat 21:00      -6.085150
00:00         -11.693898
03:00         -12.111220
Sat 06:00     -13.757591
Sun 06:00     -18.033267

[71 rows x 1 columns]
```

- The coefficients make sense!
- If it's Saturday 09:00 or Wednesday 06:00, the model is likely to predict bigger number for rentals.
- If it's Midnight or 03:00 or Sunday 06:00, the model is likely to predict smaller number for rentals.

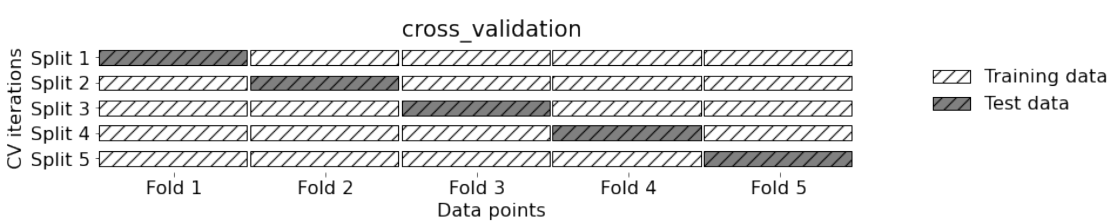
1.3.4 Cross-validation

What about cross-validation?

- We can't do regular cross-validation if we don't want to be predicting the past.
- If you carry out regular cross-validation, you'll be predicting the past given future which is not a realistic scenario for the deployment data.

[49]:

```python
mglearn.plots.plot_cross_validation()
```



There is `TimeSeriesSplit` for time series data.

[50]:

```python
from sklearn.model_selection import TimeSeriesSplit
```

[51]:

```python
X_toy = np.arange(0, 300, 10).reshape(-1, 2)
pd.DataFrame(X_toy)
```

[51]:

	0	1
0	0	10
1	20	30
2	40	50
3	60	70
4	80	90
5	100	110
6	120	130
7	140	150
8	160	170
9	180	190
10	200	210
11	220	230
12	240	250
13	260	270
14	280	290

[52]:
```
tscv = TimeSeriesSplit(n_splits=4)
print("X_toy.shape:", X_toy.shape, "\n")
for train_idx, test_idx in tscv.split(X_toy):
    print("train_idx", train_idx)
    print("test_idx ", test_idx, "\n")
```

```
X_toy.shape: (15, 2)

train_idx [0 1 2]
test_idx  [3 4 5]

train_idx [0 1 2 3 4 5]
test_idx  [6 7 8]

train_idx [0 1 2 3 4 5 6 7 8]
test_idx  [ 9 10 11]

train_idx [ 0  1  2  3  4  5  6  7  8  9 10 11]
test_idx  [12 13 14]
```

Let's try it out with Ridge on the cities data.

[53]:
```
lr = Ridge()
```

[54]:
```
scores = cross_validate(
    lr, X_hour_week_onehot_poly, y, cv=TimeSeriesSplit(),␣
 ↪return_train_score=True
)
pd.DataFrame(scores)
```

[54]:

	fit_time	score_time	test_score	train_score
0	0.001307	0.000453	0.642676	0.873182
1	0.000588	0.000244	0.828405	0.874305
2	0.000705	0.000237	0.773851	0.901262
3	0.000749	0.000287	0.696712	0.889429
4	0.002546	0.000489	0.892733	0.863889

1.4 A more complicated dataset

Rain in Australia dataset. Predicting whether or not it will rain tomorrow based on today's measurements.

[55]:
```
rain_df = pd.read_csv("data/weatherAUS.csv")
rain_df.head()
```

[55]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	\
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	

	WindGustDir	WindGustSpeed	WindDir9am	…	Humidity9am	Humidity3pm	\
0	W	44.0	W	…	71.0	22.0	
1	WNW	44.0	NNW	…	44.0	25.0	
2	WSW	46.0	W	…	38.0	30.0	
3	NE	24.0	SE	…	45.0	16.0	
4	W	41.0	ENE	…	82.0	33.0	

	Pressure9am	Pressure3pm	Cloud9am	Cloud3pm	Temp9am	Temp3pm	RainToday	\
0	1007.7	1007.1	8.0	NaN	16.9	21.8	No	
1	1010.6	1007.8	NaN	NaN	17.2	24.3	No	
2	1007.6	1008.7	NaN	2.0	21.0	23.2	No	
3	1017.6	1012.8	NaN	NaN	18.1	26.5	No	
4	1010.8	1006.0	7.0	8.0	17.8	29.7	No	

	RainTomorrow
0	No
1	No
2	No
3	No
4	No

[5 rows x 23 columns]

[56]:
```
rain_df.shape
```

[56]: (145460, 23)

Goals

- Can the date/time features help us predict the target value?
- Can we **forecast** into the future?

[57]:
```
rain_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145460 entries, 0 to 145459
Data columns (total 23 columns):
 #   Column         Non-Null Count   Dtype
---  ------         --------------   -----
 0   Date           145460 non-null  object
 1   Location       145460 non-null  object
 2   MinTemp        143975 non-null  float64
 3   MaxTemp        144199 non-null  float64
 4   Rainfall       142199 non-null  float64
 5   Evaporation    82670 non-null   float64
 6   Sunshine       75625 non-null   float64
 7   WindGustDir    135134 non-null  object
 8   WindGustSpeed  135197 non-null  float64
 9   WindDir9am     134894 non-null  object
 10  WindDir3pm     141232 non-null  object
 11  WindSpeed9am   143693 non-null  float64
 12  WindSpeed3pm   142398 non-null  float64
 13  Humidity9am    142806 non-null  float64
 14  Humidity3pm    140953 non-null  float64
 15  Pressure9am    130395 non-null  float64
 16  Pressure3pm    130432 non-null  float64
 17  Cloud9am       89572 non-null   float64
 18  Cloud3pm       86102 non-null   float64
 19  Temp9am        143693 non-null  float64
 20  Temp3pm        141851 non-null  float64
 21  RainToday      142199 non-null  object
 22  RainTomorrow   142193 non-null  object
dtypes: float64(16), object(7)
memory usage: 25.5+ MB
```

[58]:
```
rain_df.describe(include="all")
```

[58]:
```
              Date  Location        MinTemp        MaxTemp       Rainfall  \
count       145460    145460  143975.000000  144199.000000  142199.000000
unique        3436        49            NaN            NaN            NaN
top     2013-11-12  Canberra            NaN            NaN            NaN
freq            49      3436            NaN            NaN            NaN
mean           NaN       NaN      12.194034      23.221348       2.360918
std            NaN       NaN       6.398495       7.119049       8.478060
min            NaN       NaN      -8.500000      -4.800000       0.000000
```

```
25%           NaN       NaN      7.600000      17.900000       0.000000
50%           NaN       NaN     12.000000      22.600000       0.000000
75%           NaN       NaN     16.900000      28.200000       0.800000
max           NaN       NaN     33.900000      48.100000     371.000000

         Evaporation      Sunshine WindGustDir  WindGustSpeed WindDir9am  …  \
count   82670.000000  75625.000000      135134  135197.000000     134894  …
unique           NaN           NaN          16            NaN         16  …
top              NaN           NaN           W            NaN          N  …
freq             NaN           NaN        9915            NaN      11758  …
mean        5.468232      7.611178         NaN      40.035230        NaN  …
std         4.193704      3.785483         NaN      13.607062        NaN  …
min         0.000000      0.000000         NaN       6.000000        NaN  …
25%         2.600000      4.800000         NaN      31.000000        NaN  …
50%         4.800000      8.400000         NaN      39.000000        NaN  …
75%         7.400000     10.600000         NaN      48.000000        NaN  …
max       145.000000     14.500000         NaN     135.000000        NaN  …

          Humidity9am    Humidity3pm   Pressure9am    Pressure3pm  \
count   142806.000000  140953.000000  130395.00000  130432.000000
unique            NaN            NaN           NaN            NaN
top               NaN            NaN           NaN            NaN
freq              NaN            NaN           NaN            NaN
mean        68.880831      51.539116    1017.64994    1015.255889
std         19.029164      20.795902       7.10653       7.037414
min          0.000000       0.000000     980.50000     977.100000
25%         57.000000      37.000000    1012.90000    1010.400000
50%         70.000000      52.000000    1017.60000    1015.200000
75%         83.000000      66.000000    1022.40000    1020.000000
max        100.000000     100.000000    1041.00000    1039.600000

            Cloud9am      Cloud3pm        Temp9am       Temp3pm  RainToday  \
count   89572.000000  86102.000000  143693.000000  141851.00000     142199
unique           NaN           NaN            NaN           NaN          2
top              NaN           NaN            NaN           NaN         No
freq             NaN           NaN            NaN           NaN     110319
mean        4.447461      4.509930      16.990631      21.68339        NaN
std         2.887159      2.720357       6.488753       6.93665        NaN
min         0.000000      0.000000      -7.200000      -5.40000        NaN
25%         1.000000      2.000000      12.300000      16.60000        NaN
50%         5.000000      5.000000      16.700000      21.10000        NaN
75%         7.000000      7.000000      21.600000      26.40000        NaN
max         9.000000      9.000000      40.200000      46.70000        NaN

        RainTomorrow
count         142193
unique             2
```

```
top                No
freq           110316
mean              NaN
std               NaN
min               NaN
25%               NaN
50%               NaN
75%               NaN
max               NaN

[11 rows x 23 columns]
```

- A number of missing values.
- Some target values are also missing. I'm dropping these rows.

[59]:
```
rain_df = rain_df[rain_df["RainTomorrow"].notna()]
rain_df.shape
```

[59]:
```
(142193, 23)
```

Parsing datetimes

- In general, datetimes are a huge pain! Think of all the formats: MM-DD-YY, DD-MM-YY, YY-MM-DD, MM/DD/YY, DD/MM/YY, DD/MM/YYYY, 20:45, 8:45am, 8:45 PM, 8:45a, 08:00, 8:10:20, …….
- No, seriously, dealing with datetimes is THE WORST.
  - Time zones.
  - Daylight savings…
- Thankfully, pandas does a pretty good job here.

[60]:
```
dates_rain = pd.to_datetime(rain_df["Date"])
dates_rain
```

[60]:
```
0         2008-12-01
1         2008-12-02
2         2008-12-03
3         2008-12-04
4         2008-12-05
             …
145454    2017-06-20
145455    2017-06-21
145456    2017-06-22
145457    2017-06-23
145458    2017-06-24
Name: Date, Length: 142193, dtype: datetime64[ns]
```

They are all the same format, so we can also compare dates:

[61]:
```
dates_rain[1] - dates_rain[0]
```

[61]: Timedelta('1 days 00:00:00')

[62]:
```
dates_rain[1] > dates_rain[0]
```

[62]: True

[63]:
```
(dates_rain[1] - dates_rain[0]).total_seconds()
```

[63]: 86400.0

We can also easily extract information from the date columns.

[64]:
```
dates_rain[1]
```

[64]: Timestamp('2008-12-02 00:00:00')

[65]:
```
dates_rain[1].month_name()
```

[65]: 'December'

[66]:
```
dates_rain[1].day_name()
```

[66]: 'Tuesday'

[67]:
```
dates_rain[1].is_year_end
```

[67]: False

[68]:
```
dates_rain[1].is_leap_year
```

[68]: True

[69]:
```
dates_rain[dates_rain.map(lambda d: d.is_year_end and d.is_leap_year)].unique()
```

[69]:
```
array(['2008-12-31T00:00:00.000000000', '2016-12-31T00:00:00.000000000'],
      dtype='datetime64[ns]')
```

Above pandas identified the date column automatically. You can tell pandas to parse the dates when reading in the CSV:

[70]:
```
rain_df = pd.read_csv("data/weatherAUS.csv", parse_dates=["Date"])
rain_df.head()
```

[70]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	\
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	

```
  WindGustDir  WindGustSpeed WindDir9am  … Humidity9am  Humidity3pm  \
0           W           44.0          W  …        71.0         22.0
1         WNW           44.0        NNW  …        44.0         25.0
2         WSW           46.0          W  …        38.0         30.0
3          NE           24.0         SE  …        45.0         16.0
4           W           41.0        ENE  …        82.0         33.0

   Pressure9am  Pressure3pm  Cloud9am  Cloud3pm  Temp9am  Temp3pm  RainToday  \
0       1007.7       1007.1       8.0       NaN     16.9     21.8         No
1       1010.6       1007.8       NaN       NaN     17.2     24.3         No
2       1007.6       1008.7       NaN       2.0     21.0     23.2         No
3       1017.6       1012.8       NaN       NaN     18.1     26.5         No
4       1010.8       1006.0       7.0       8.0     17.8     29.7         No

   RainTomorrow
0            No
1            No
2            No
3            No
4            No

[5 rows x 23 columns]
```

[71]:
```python
rain_df["RainTomorrow"].isna().sum()
```

[71]:
```
3267
```

[72]:
```python
rain_df = rain_df[rain_df["RainTomorrow"].notna()]
rain_df.shape
```

[72]:
```
(142193, 23)
```

[73]:
```python
rain_df["Date"].head()
```

[73]:
```
0   2008-12-01
1   2008-12-02
2   2008-12-03
3   2008-12-04
4   2008-12-05
Name: Date, dtype: datetime64[ns]
```

[74]:
```python
rain_df["Date"].unique().shape
```

[74]:
```
(3436,)
```

1.5 Train/test splits

- Remember that we should not be calling the usual `train_test_split` with shuffling because

- If we want to forecast, we aren't allowed to know what happened in the future!

[75]:
```
rain_df["Date"].min()
```

[75]: Timestamp('2007-11-01 00:00:00')

[76]:
```
rain_df["Date"].max()
```

[76]: Timestamp('2017-06-25 00:00:00')

- It looks like we have 10 years of data.
- Let's use the last 2 years for test.

[77]:
```
train_df = rain_df.query("Date <= 20150630")
test_df = rain_df.query("Date >  20150630")
```

[78]:
```
len(train_df)
```

[78]: 107502

[79]:
```
len(test_df)
```

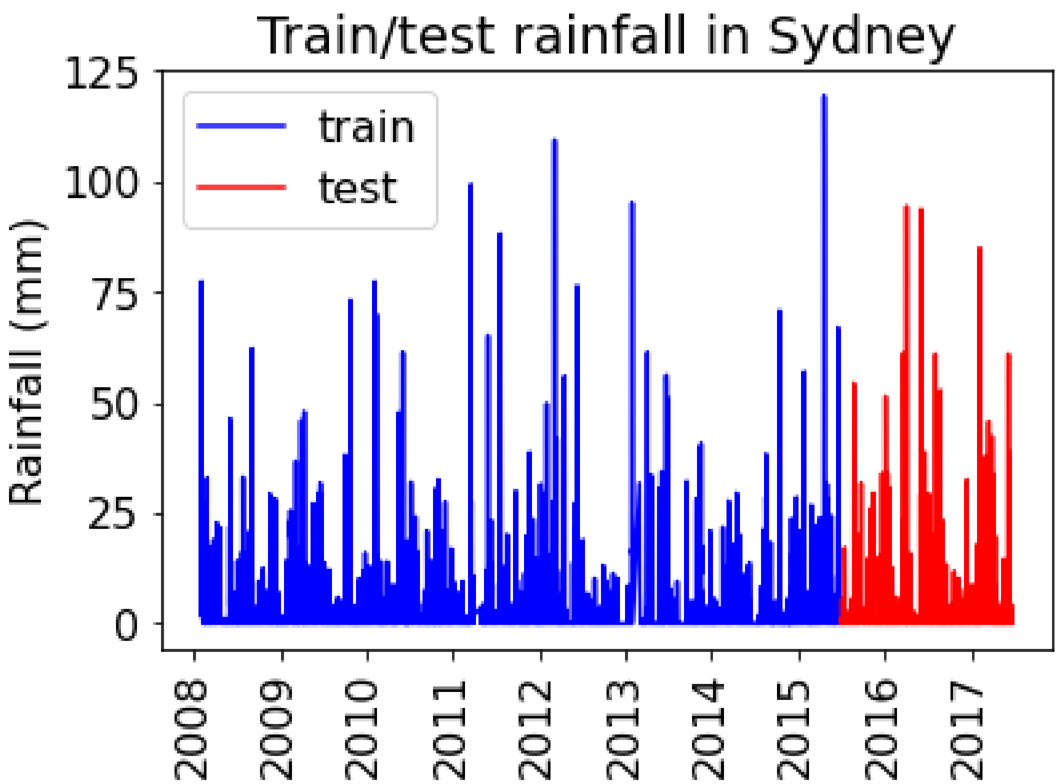
[79]: 34691

[80]:
```
len(test_df) / (len(train_df) + len(test_df))
```

[80]: 0.24397122221206388

As we can see, we're still using about 25% of our data as test data.

[81]:
```
train_df_sort = train_df.query("Location == 'Sydney'").sort_values(by="Date")
test_df_sort = test_df.query("Location == 'Sydney'").sort_values(by="Date")

plt.plot(train_df_sort["Date"], train_df_sort["Rainfall"], "b", label="train")
plt.plot(test_df_sort["Date"], test_df_sort["Rainfall"], "r", label="test")
plt.xticks(rotation="vertical")
plt.legend()
plt.ylabel("Rainfall (mm)")
plt.title("Train/test rainfall in Sydney");
```



We're learning relationships from the blue part; predicting only using features in the red part from the day before.

Let's define a preprocessor with a column transformer.

[82]:
```
train_df.columns
```

[82]:
```
Index(['Date', 'Location', 'MinTemp', 'MaxTemp', 'Rainfall', 'Evaporation',
       'Sunshine', 'WindGustDir', 'WindGustSpeed', 'WindDir9am', 'WindDir3pm',
       'WindSpeed9am', 'WindSpeed3pm', 'Humidity9am', 'Humidity3pm',
       'Pressure9am', 'Pressure3pm', 'Cloud9am', 'Cloud3pm', 'Temp9am',
       'Temp3pm', 'RainToday', 'RainTomorrow'],
      dtype='object')
```

[83]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 107502 entries, 0 to 144733
Data columns (total 23 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   Date            107502 non-null  datetime64[ns]
 1   Location        107502 non-null  object
```

```
 2   MinTemp        107050 non-null  float64
 3   MaxTemp        107292 non-null  float64
 4   Rainfall       106424 non-null  float64
 5   Evaporation    66221 non-null   float64
 6   Sunshine       62320 non-null   float64
 7   WindGustDir    100103 non-null  object
 8   WindGustSpeed  100146 non-null  float64
 9   WindDir9am     99515 non-null   object
 10  WindDir3pm     105314 non-null  object
 11  WindSpeed9am   106322 non-null  float64
 12  WindSpeed3pm   106319 non-null  float64
 13  Humidity9am    106112 non-null  float64
 14  Humidity3pm    106180 non-null  float64
 15  Pressure9am    97217 non-null   float64
 16  Pressure3pm    97253 non-null   float64
 17  Cloud9am       68523 non-null   float64
 18  Cloud3pm       67501 non-null   float64
 19  Temp9am        106705 non-null  float64
 20  Temp3pm        106816 non-null  float64
 21  RainToday      106424 non-null  object
 22  RainTomorrow   107502 non-null  object
dtypes: datetime64[ns](1), float64(16), object(6)
memory usage: 19.7+ MB
```

- We have missing data.
- We have categorical features and numeric features.

- Let's define feature types.
- Let's start with **dropping the date** column and treating it as a **usual supervised machine** learning problem.

[84]:
```
numeric_features = [
    "MinTemp",
    "MaxTemp",
    "Rainfall",
    "Evaporation",
    "Sunshine",
    "WindGustSpeed",
    "WindSpeed9am",
    "WindSpeed3pm",
    "Humidity9am",
    "Humidity3pm",
    "Pressure9am",
    "Pressure3pm",
    "Cloud9am",
    "Cloud3pm",
    "Temp9am",
    "Temp3pm",
```

```
]
categorical_features = [
    "Location",
    "WindGustDir",
    "WindDir9am",
    "WindDir3pm",
    "RainToday",
]
drop_features = [
    "Date",
    "RainTomorrow",
]
```

[85]:
```
def preprocess_features(
    train_df,
    test_df,
    numeric_features,
    categorical_features,
    drop_features,
):

    all_features = set(numeric_features + categorical_features + drop_features)
    if set(train_df.columns) != all_features:
        print("Missing columns", set(train_df.columns) - all_features)
        print("Extra columns", all_features - set(train_df.columns))
        raise Exception("Columns do not match")

    numeric_transformer = make_pipeline(
        SimpleImputer(strategy="median"), StandardScaler()
    )
    categorical_transformer = make_pipeline(
        SimpleImputer(strategy="constant", fill_value="?"),
        OneHotEncoder(handle_unknown="ignore", sparse=False),
    )

    preprocessor = make_column_transformer(
        (numeric_transformer, numeric_features),
        (categorical_transformer, categorical_features),
        ("drop", drop_features),
    )
    preprocessor.fit(train_df)
    ohe_feature_names = (
        preprocessor.named_transformers_["pipeline-2"]
        .named_steps["onehotencoder"]
        .get_feature_names_out()
        .tolist()
    )
```

```python
    new_columns = numeric_features + ohe_feature_names

    X_train_enc = pd.DataFrame(
        preprocessor.transform(train_df), index=train_df.index,␣
↪columns=new_columns
    )
    X_test_enc = pd.DataFrame(
        preprocessor.transform(test_df), index=test_df.index,␣
↪columns=new_columns
    )

    y_train = train_df["RainTomorrow"]
    y_test = test_df["RainTomorrow"]

    return X_train_enc, y_train, X_test_enc, y_test, preprocessor
```

[86]:
```python
X_train_enc, y_train, X_test_enc, y_test, preprocessor = preprocess_features(
    train_df,
    test_df,
    numeric_features,
    categorical_features,
    drop_features,
)
```

[87]:
```python
X_train_enc.head()
```

[87]:
```
    MinTemp   MaxTemp  Rainfall  Evaporation  Sunshine  WindGustSpeed  \
0  0.204302 -0.027112 -0.205323    -0.140641  0.160729       0.298612
1 -0.741037  0.287031 -0.275008    -0.140641  0.160729       0.298612
2  0.125523  0.372706 -0.275008    -0.140641  0.160729       0.450132
3 -0.457435  0.701128 -0.275008    -0.140641  0.160729      -1.216596
4  0.850283  1.315134 -0.158867    -0.140641  0.160729       0.071330

   WindSpeed9am  WindSpeed3pm  Humidity9am  Humidity3pm  …  x3_SE  x3_SSE  \
0      0.666166      0.599894     0.115428    -1.433514  …    0.0     0.0
1     -1.125617      0.373275    -1.314929    -1.288002  …    0.0     0.0
2      0.554180      0.826513    -1.632786    -1.045481  …    0.0     0.0
3     -0.341712     -1.099749    -1.261953    -1.724539  …    0.0     0.0
4     -0.789657      0.146656     0.698167    -0.899969  …    0.0     0.0

   x3_SSW  x3_SW  x3_W  x3_WNW  x3_WSW  x4_?  x4_No  x4_Yes
0     0.0    0.0   0.0     1.0     0.0   0.0    1.0     0.0
1     0.0    0.0   0.0     0.0     1.0   0.0    1.0     0.0
2     0.0    0.0   0.0     0.0     1.0   0.0    1.0     0.0
3     0.0    0.0   0.0     0.0     0.0   0.0    1.0     0.0
4     0.0    0.0   0.0     0.0     0.0   0.0    1.0     0.0
```

```
[5 rows x 119 columns]
```

1.5.1 DummyClassifier

[88]:
```python
y_train.value_counts().to_frame().assign(ratio=y_train.
    ↪value_counts(normalize=True))
```

[88]:

	RainTomorrow	ratio
No	83320	0.775055
Yes	24182	0.224945

[89]:
```python
dc = DummyClassifier(strategy="prior")
dc.fit(train_df, y_train);
```

[90]:
```python
dc.score(train_df, y_train)
```

[90]: 0.7750553478074826

[91]:
```python
dc.score(test_df, y_test)
```

[91]: 0.7781845435415525

1.5.2 LogisticRegression

The function below trains a logistic regression model on the train set, reports train and test scores, and returns learned coefficients as a dataframe.

[92]:
```python
def score_lr_print_coeff(preprocessor, train_df, y_train, test_df, y_test,␣
    ↪columns):
    lr_pipe = make_pipeline(preprocessor, LogisticRegression(max_iter=1000))
    lr_pipe.fit(train_df, y_train)
    print("Train score: {:.2f}".format(lr_pipe.score(train_df, y_train)))
    print("Test score: {:.2f}".format(lr_pipe.score(test_df, y_test)))
    lr_coef = pd.DataFrame(
        data=lr_pipe.named_steps["logisticregression"].coef_.flatten(),
        index=columns,
        columns=["Coef"],
    )
    return lr_coef.sort_values(by="Coef", ascending=False)
```

[93]:
```python
score_lr_print_coeff(preprocessor, train_df, y_train, test_df, y_test,␣
    ↪X_train_enc.columns)
```

```
Train score: 0.85
Test score: 0.84
```

[93]:

	Coef
Humidity3pm	1.243181

```
x4_?             0.924002
Pressure9am      0.865490
x0_Witchcliffe   0.729016
WindGustSpeed    0.720411
…                     …
x0_Townsville   -0.718907
x0_Katherine    -0.725970
x0_Wollongong   -0.749053
x0_MountGinini  -0.965140
Pressure3pm     -1.221811

[119 rows x 1 columns]
```

1.5.3 Cross-validation

- We can carry out cross-validation using TimeSeriesSplit.
- However, things are actually more complicated here because this dataset has **multiple time series**, one per location.

[94]:
```
train_df
```

[94]:
```
              Date Location  MinTemp  MaxTemp  Rainfall  Evaporation  Sunshine  \
0       2008-12-01   Albury     13.4     22.9       0.6          NaN       NaN
1       2008-12-02   Albury      7.4     25.1       0.0          NaN       NaN
2       2008-12-03   Albury     12.9     25.7       0.0          NaN       NaN
3       2008-12-04   Albury      9.2     28.0       0.0          NaN       NaN
4       2008-12-05   Albury     17.5     32.3       1.0          NaN       NaN
…              …        …       …        …         …          …
144729  2015-06-26    Uluru      3.8     18.3       0.0          NaN       NaN
144730  2015-06-27    Uluru      2.5     17.1       0.0          NaN       NaN
144731  2015-06-28    Uluru      4.5     19.6       0.0          NaN       NaN
144732  2015-06-29    Uluru      7.6     22.0       0.0          NaN       NaN
144733  2015-06-30    Uluru      6.8     21.1       0.0          NaN       NaN

       WindGustDir  WindGustSpeed WindDir9am  … Humidity9am  Humidity3pm  \
0                W           44.0          W  …        71.0         22.0
1              WNW           44.0        NNW  …        44.0         25.0
2              WSW           46.0          W  …        38.0         30.0
3               NE           24.0         SE  …        45.0         16.0
4                W           41.0        ENE  …        82.0         33.0
…              …            …         …  …         …            …
144729           E           39.0        ESE  …        73.0         37.0
144730           E           41.0        ESE  …        69.0         40.0
144731         ENE           35.0        ESE  …        69.0         39.0
144732         ESE           33.0         SE  …        67.0         37.0
144733         ESE           35.0        ESE  …        81.0         35.0

        Pressure9am  Pressure3pm  Cloud9am  Cloud3pm  Temp9am  Temp3pm  \
```

```
0              1007.7       1007.1       8.0       NaN     16.9      21.8
1              1010.6       1007.8       NaN       NaN     17.2      24.3
2              1007.6       1008.7       NaN       2.0     21.0      23.2
3              1017.6       1012.8       NaN       NaN     18.1      26.5
4              1010.8       1006.0       7.0       8.0     17.8      29.7
…                 …            …       …      …       …       …
144729         1031.5       1027.6       NaN       NaN      8.8      17.2
144730         1029.9       1026.0       NaN       NaN      7.0      15.7
144731         1028.7       1025.0       NaN       3.0      8.9      18.0
144732         1027.2       1023.8       6.0       7.0     11.7      21.5
144733         1028.6       1025.2       3.0       NaN     10.6      20.2

        RainToday  RainTomorrow
0              No            No
1              No            No
2              No            No
3              No            No
4              No            No
…            …            …
144729         No            No
144730         No            No
144731         No            No
144732         No            No
144733         No            No

[107502 rows x 23 columns]
```

[95]:
```python
train_df.groupby(["Date", "Location"]).size().unique()
```

[95]:
```
array([1])
```

[96]:
```python
train_df.groupby("Location").size().unique()
```

[96]:
```
array([2369, 2305, 2290, 2314, 2226, 2304, 2444, 2310, 2696, 2277, 2229,
       2219, 2467, 2289, 2463,  847, 2306, 2013, 2283, 2281, 2165, 2182,
       2255,  845, 2220, 2282, 2280, 2048, 2252, 2274, 2228, 2235, 2611,
       2313, 2279,  807, 2118, 1888, 2240, 2267])
```

[97]:
```python
train_df.groupby("Date").size().unique()
```

[97]:
```
array([ 1,  2,  8,  7, 22, 23, 45, 44, 46, 43, 42, 39, 41, 48, 49, 47, 40])
```

[98]:
```python
train_df.sort_values(by=["Date"])
```

[98]:
```
             Date          Location  MinTemp  MaxTemp  Rainfall  Evaporation  \
45587  2007-11-01          Canberra      8.0     24.3       0.0          3.4
45588  2007-11-02          Canberra     14.0     26.9       3.6          4.4
45589  2007-11-03          Canberra     13.7     23.4       3.6          5.8
```

```
45590  2007-11-04          Canberra     13.3     15.5      39.8          7.2
45591  2007-11-05          Canberra      7.6     16.1       2.8          5.6
…          …                  …      …     …       …          …
57415  2015-06-30          Ballarat     -0.3     10.5       0.0          NaN
119911 2015-06-30      PerthAirport     10.1     23.5       0.0          3.2
60455  2015-06-30           Bendigo      0.3     11.4       0.0          NaN
66473  2015-06-30  MelbourneAirport      3.2     13.2       0.0          0.8
144733 2015-06-30             Uluru      6.8     21.1       0.0          NaN

        Sunshine WindGustDir  WindGustSpeed WindDir9am  … Humidity9am  \
45587        6.3          NW           30.0         SW  …        68.0
45588        9.7         ENE           39.0          E  …        80.0
45589        3.3          NW           85.0          N  …        82.0
45590        9.1          NW           54.0        WNW  …        62.0
45591       10.6         SSE           50.0        SSE  …        68.0
…          …        …            …         …  …         …
57415        NaN           S           26.0        NaN  …        99.0
119911       5.8         NNE           31.0         NE  …        48.0
60455        NaN           W           19.0        NaN  …        89.0
66473        3.9           N           20.0          N  …        91.0
144733       NaN         ESE           35.0        ESE  …        81.0

        Humidity3pm  Pressure9am  Pressure3pm  Cloud9am  Cloud3pm  Temp9am  \
45587          29.0       1019.7       1015.0       7.0       7.0     14.4
45588          36.0       1012.4       1008.4       5.0       3.0     17.5
45589          69.0       1009.5       1007.2       8.0       7.0     15.4
45590          56.0       1005.5       1007.0       2.0       7.0     13.5
45591          49.0       1018.3       1018.5       7.0       7.0     11.1
…             …           …          …          …        …       …
57415          63.0       1029.5       1027.7       NaN       8.0      4.7
119911         33.0       1023.6       1021.7       7.0       6.0     13.3
60455          56.0       1029.3       1027.4       8.0       7.0      6.4
66473          50.0       1029.6       1027.3       2.0       7.0      5.3
144733         35.0       1028.6       1025.2       3.0       NaN     10.6

        Temp3pm  RainToday  RainTomorrow
45587      23.6         No           Yes
45588      25.7        Yes           Yes
45589      20.2        Yes           Yes
45590      14.1        Yes           Yes
45591      15.4        Yes            No
…         …         …            …
57415       9.3         No            No
119911     22.2         No            No
60455      10.5         No            No
66473      11.9         No            No
144733     20.2         No            No
```

```
[107502 rows x 23 columns]
```

- It seems the dataframe is sorted by location, and then time.
- Our first approach will be to ignore the fact that we have multiple time series and just **OHE the location**
- We'll have **multiple measurements for a given timestamp**, and that's OK.
- But, `TimeSeriesSplit` expects the dataframe to be sorted by date so…

[99]:
```
train_df_ordered = train_df.sort_values(by=["Date"])
y_train_ordered = train_df_ordered["RainTomorrow"]
```

[100]:
```
train_df_ordered
```

[100]:
```
              Date          Location  MinTemp  MaxTemp  Rainfall  Evaporation  \
45587   2007-11-01          Canberra      8.0     24.3       0.0          3.4
45588   2007-11-02          Canberra     14.0     26.9       3.6          4.4
45589   2007-11-03          Canberra     13.7     23.4       3.6          5.8
45590   2007-11-04          Canberra     13.3     15.5      39.8          7.2
45591   2007-11-05          Canberra      7.6     16.1       2.8          5.6
…              …                 …      …        …        …         …
57415   2015-06-30          Ballarat     -0.3     10.5       0.0          NaN
119911  2015-06-30      PerthAirport     10.1     23.5       0.0          3.2
60455   2015-06-30           Bendigo      0.3     11.4       0.0          NaN
66473   2015-06-30  MelbourneAirport      3.2     13.2       0.0          0.8
144733  2015-06-30             Uluru      6.8     21.1       0.0          NaN

        Sunshine WindGustDir  WindGustSpeed WindDir9am  … Humidity9am  \
45587        6.3          NW           30.0         SW  …        68.0
45588        9.7         ENE           39.0          E  …        80.0
45589        3.3          NW           85.0          N  …        82.0
45590        9.1          NW           54.0        WNW  …        62.0
45591       10.6         SSE           50.0        SSE  …        68.0
…              …         …              …        …  …           …
57415        NaN           S           26.0        NaN  …        99.0
119911       5.8         NNE           31.0         NE  …        48.0
60455        NaN           W           19.0        NaN  …        89.0
66473        3.9           N           20.0          N  …        91.0
144733       NaN         ESE           35.0        ESE  …        81.0

        Humidity3pm  Pressure9am  Pressure3pm  Cloud9am  Cloud3pm  Temp9am  \
45587          29.0       1019.7       1015.0       7.0       7.0     14.4
45588          36.0       1012.4       1008.4       5.0       3.0     17.5
45589          69.0       1009.5       1007.2       8.0       7.0     15.4
45590          56.0       1005.5       1007.0       2.0       7.0     13.5
45591          49.0       1018.3       1018.5       7.0       7.0     11.1
…                 …          …          …         …        …
57415          63.0       1029.5       1027.7       NaN       8.0      4.7
```

```
119911          33.0       1023.6       1021.7       7.0        6.0      13.3
60455           56.0       1029.3       1027.4       8.0        7.0       6.4
66473           50.0       1029.6       1027.3       2.0        7.0       5.3
144733          35.0       1028.6       1025.2       3.0        NaN      10.6

        Temp3pm  RainToday  RainTomorrow
45587      23.6         No           Yes
45588      25.7        Yes           Yes
45589      20.2        Yes           Yes
45590      14.1        Yes           Yes
45591      15.4        Yes            No
…          …         …            …
57415       9.3         No            No
119911     22.2         No            No
60455      10.5         No            No
66473      11.9         No            No
144733     20.2         No            No

[107502 rows x 23 columns]
```

[101]:
```python
lr_pipe = make_pipeline(preprocessor, LogisticRegression(max_iter=1000))
cross_val_score(lr_pipe, train_df_ordered, y_train_ordered,␣
 ↪cv=TimeSeriesSplit()).mean()
```

[101]: 0.8478874811631412

[102]:
```python
train_df_sydney = train_df.query("Location == 'Sydney'").sort_values(by="Date")
cross_val_score(
    lr_pipe, train_df_sydney, train_df_sydney["RainTomorrow"],␣
 ↪cv=TimeSeriesSplit()).mean()
```

[102]: 0.8317241379310344

[103]:
```python
def cross_val_score_loc(train_df_loc):
    # print(train_df_loc['Location'].unique())
    return cross_val_score(
        lr_pipe, train_df_loc, train_df_loc["RainTomorrow"],␣
 ↪cv=TimeSeriesSplit()).mean()

location_results = train_df.groupby("Location").apply(cross_val_score_loc)
location_results.head(3)
```

[103]:
```
Location
Adelaide    0.862944
Albany      0.804688
Albury      0.875066
dtype: float64
```

[104]:
```
location_results.describe()
```

[104]:
```
count    49.000000
mean      0.853753
std       0.043287
min       0.776316
25%       0.813008
50%       0.856511
75%       0.884156
max       0.943536
dtype: float64
```

[105]:
```
location_results.rename("CV Score").sort_values(ascending=False).to_frame().
↪reset_index()
```

[105]:

	Location	CV Score
0	Woomera	0.943536
1	AliceSprings	0.934026
2	Uluru	0.928358
3	Mildura	0.921579
4	Perth	0.912895
5	Cobar	0.908707
6	PearceRAAF	0.908504
7	Moree	0.906667
8	PerthAirport	0.903158
9	Katherine	0.892199
10	WaggaWagga	0.890000
11	Townsville	0.888312
12	Bendigo	0.884156
13	Tuggeranong	0.880739
14	SalmonGums	0.879032
15	Albury	0.875066
16	Nhil	0.874286
17	Nuriootpa	0.872632
18	Richmond	0.872237
19	Witchcliffe	0.865416
20	BadgerysCreek	0.864151
21	Adelaide	0.862944
22	Canberra	0.862361
23	Penrith	0.861333
24	Brisbane	0.856511
25	Darwin	0.854501
26	GoldCoast	0.845144
27	Launceston	0.842708
28	Ballarat	0.839063
29	MelbourneAirport	0.833158
30	Sydney	0.831724

```
31      MountGambier  0.827013
32     SydneyAirport  0.824737
33          Watsonia  0.820000
34        Wollongong  0.817507
35         Melbourne  0.813134
36          Dartmoor  0.813008
37         NorahHead  0.809730
38           Walpole  0.809065
39              Sale  0.808947
40       Williamtown  0.808917
41            Hobart  0.808780
42            Cairns  0.807792
43      CoffsHarbour  0.805930
44            Albany  0.804688
45       MountGinini  0.803857
46         Newcastle  0.795733
47          Portland  0.783641
48     NorfolkIsland  0.776316
```

Just as practice, for now, let's ignore location and just consider date.

1.6 Encoding date/time as feature(s)

- Can we use the `Date` to help us predict the target?
- Probably! E.g. different amounts of rain in different seasons.
- This is feature engineering!

1.6.1 Encoding time as an number

- Idea 1: create a column of "days since Nov 1, 2007" which is the first day in the dataset.

[106]:
```python
train_df["Date"].min()
```

[106]:
```
Timestamp('2007-11-01 00:00:00')
```

[107]:
```python
train_df = rain_df.query("Date <= 20150630")
test_df = rain_df.query("Date >  20150630")
```

[108]:
```python
first_day = train_df["Date"].min()

train_df = train_df.assign(
    Days_since=train_df["Date"].apply(lambda x: (x - first_day).days)
)
test_df = test_df.assign(
    Days_since=test_df["Date"].apply(lambda x: (x - first_day).days)
)
```

[109]:
```python
train_df.sort_values(by="Date").head()
```

[109]:

```
             Date  Location  MinTemp  MaxTemp  Rainfall  Evaporation  Sunshine  \
45587  2007-11-01  Canberra      8.0     24.3       0.0          3.4       6.3
45588  2007-11-02  Canberra     14.0     26.9       3.6          4.4       9.7
45589  2007-11-03  Canberra     13.7     23.4       3.6          5.8       3.3
45590  2007-11-04  Canberra     13.3     15.5      39.8          7.2       9.1
45591  2007-11-05  Canberra      7.6     16.1       2.8          5.6      10.6

      WindGustDir  WindGustSpeed WindDir9am  … Humidity3pm  Pressure9am  \
45587          NW           30.0         SW  …        29.0       1019.7
45588         ENE           39.0          E  …        36.0       1012.4
45589          NW           85.0          N  …        69.0       1009.5
45590          NW           54.0        WNW  …        56.0       1005.5
45591         SSE           50.0        SSE  …        49.0       1018.3

       Pressure3pm  Cloud9am  Cloud3pm  Temp9am  Temp3pm  RainToday  \
45587       1015.0       7.0       7.0     14.4     23.6         No
45588       1008.4       5.0       3.0     17.5     25.7        Yes
45589       1007.2       8.0       7.0     15.4     20.2        Yes
45590       1007.0       2.0       7.0     13.5     14.1        Yes
45591       1018.5       7.0       7.0     11.1     15.4        Yes

       RainTomorrow  Days_since
45587           Yes           0
45588           Yes           1
45589           Yes           2
45590           Yes           3
45591            No           4

[5 rows x 24 columns]
```

[110]:

```python
X_train_enc, y_train, X_test_enc, y_test, preprocessor = preprocess_features(
    train_df,
    test_df,
    numeric_features + ["Days_since"],
    categorical_features,
    drop_features,
)
```

[111]:

```python
score_lr_print_coeff(preprocessor, train_df, y_train, test_df, y_test,␣
 ↪X_train_enc.columns)
```

```
Train score: 0.85
Test score: 0.84
```

[111]:

```
                   Coef
Humidity3pm    1.243092
x4_?           0.935320
Pressure9am    0.864500
```

```
x0_Witchcliffe  0.731007
WindGustSpeed   0.720028
…                    …
x0_Townsville  -0.716770
x0_Katherine   -0.738735
x0_Wollongong  -0.746094
x0_MountGinini -0.963654
Pressure3pm    -1.221826

[120 rows x 1 columns]
```

- Not much improvement in the scores
- Can you think of other ways to generate features from the `Date` column?

1.6.2 One-hot encoding of the month

- Idea 2: month
- The month seems relevant here. How should we encode the month?
- Encode it as a categorical variable?

[112]:
```
train_df = rain_df.query("Date <= 20150630")
test_df = rain_df.query("Date >  20150630")
```

[113]:
```
# use month_name() to get the actual string
train_df = train_df.assign(Month=train_df["Date"].apply(lambda x: x.
 ↪month_name()))
test_df = test_df.assign(Month=test_df["Date"].apply(lambda x: x.month_name()))
```

[114]:
```
# To ensure correct month ordering
months_cat = pd.CategoricalDtype(ordered=True,
    categories=['January', 'February', 'March', 'April', 'May', 'June',
                'July', 'August', 'September', 'October', 'November',␣
 ↪'December'])

train_df["Month"] = train_df["Month"].astype(months_cat)
test_df["Month"] = test_df["Month"].astype(months_cat)
```

[115]:
```
train_df[["Date", "Month"]].sort_values(by="Month")
```

[115]:
```
            Date     Month
32616 2015-01-04   January
38285 2014-01-20   January
38284 2014-01-19   January
38283 2014-01-18   January
38282 2014-01-17   January
…            …         …
47724 2013-12-05  December
47723 2013-12-04  December
```

```
47722 2013-12-03  December
47720 2013-12-01  December
0     2008-12-01  December

[107502 rows x 2 columns]
```

[116]:
```
X_train_enc, y_train, X_test_enc, y_test, preprocessor = preprocess_features(
    train_df, test_df, numeric_features, categorical_features + ["Month"],
 ↪drop_features
)
```

[117]:
```
score_lr_print_coeff(preprocessor, train_df, y_train, test_df, y_test,
 ↪X_train_enc.columns)
```

```
Train score: 0.85
Test score: 0.84
```

[117]:

	Coef
Humidity3pm	1.266873
x4_?	0.942135
Pressure9am	0.799146
x0_Witchcliffe	0.748923
WindGustSpeed	0.705617
…	…
x0_Darwin	-0.736057
x0_Wollongong	-0.748181
x0_Townsville	-0.903216
x0_Katherine	-0.930322
Pressure3pm	-1.182015

[131 rows x 1 columns]

1.6.3 One-hot encoding seasons

How about just summer/winter as a feature?

[118]:
```
def get_season(month):
    WINTER_MONTHS = ["June", "July", "August"]
    AUTUMN_MONTHS = ["March", "April", "May"]
    SUMMER_MONTHS = ["December", "January", "February"]
    SPRING_MONTHS = ["September", "October", "November"]
    if month in WINTER_MONTHS:
        return "Winter"
    elif month in AUTUMN_MONTHS:
        return "Autumn"
    elif month in SUMMER_MONTHS:
        return "Summer"
    else:
```

```
        return "Fall"
```

[119]:
```
train_df = train_df.assign(Season=train_df["Month"].apply(get_season))
test_df = test_df.assign(Season=test_df["Month"].apply(get_season))
```

[120]:
```
train_df
```

[120]:
```
             Date Location  MinTemp  MaxTemp  Rainfall  Evaporation  Sunshine  \
0      2008-12-01   Albury     13.4     22.9       0.6          NaN       NaN
1      2008-12-02   Albury      7.4     25.1       0.0          NaN       NaN
2      2008-12-03   Albury     12.9     25.7       0.0          NaN       NaN
3      2008-12-04   Albury      9.2     28.0       0.0          NaN       NaN
4      2008-12-05   Albury     17.5     32.3       1.0          NaN       NaN
…             …        …      …       …        …           …
144729 2015-06-26    Uluru      3.8     18.3       0.0          NaN       NaN
144730 2015-06-27    Uluru      2.5     17.1       0.0          NaN       NaN
144731 2015-06-28    Uluru      4.5     19.6       0.0          NaN       NaN
144732 2015-06-29    Uluru      7.6     22.0       0.0          NaN       NaN
144733 2015-06-30    Uluru      6.8     21.1       0.0          NaN       NaN

       WindGustDir  WindGustSpeed WindDir9am  … Pressure9am  Pressure3pm  \
0                W           44.0          W  …      1007.7       1007.1
1              WNW           44.0        NNW  …      1010.6       1007.8
2              WSW           46.0          W  …      1007.6       1008.7
3               NE           24.0         SE  …      1017.6       1012.8
4                W           41.0        ENE  …      1010.8       1006.0
…              …              …         …   …         …            …
144729           E           39.0        ESE  …      1031.5       1027.6
144730           E           41.0        ESE  …      1029.9       1026.0
144731         ENE           35.0        ESE  …      1028.7       1025.0
144732         ESE           33.0         SE  …      1027.2       1023.8
144733         ESE           35.0        ESE  …      1028.6       1025.2

        Cloud9am  Cloud3pm  Temp9am  Temp3pm  RainToday  RainTomorrow  \
0            8.0       NaN     16.9     21.8         No            No
1            NaN       NaN     17.2     24.3         No            No
2            NaN       2.0     21.0     23.2         No            No
3            NaN       NaN     18.1     26.5         No            No
4            7.0       8.0     17.8     29.7         No            No
…          …       …      …       …        …          …
144729       NaN       NaN      8.8     17.2         No            No
144730       NaN       NaN      7.0     15.7         No            No
144731       NaN       3.0      8.9     18.0         No            No
144732       6.0       7.0     11.7     21.5         No            No
144733       3.0       NaN     10.6     20.2         No            No

           Month  Season
```

```
0       December  Summer
1       December  Summer
2       December  Summer
3       December  Summer
4       December  Summer
…            …      …
144729      June  Winter
144730      June  Winter
144731      June  Winter
144732      June  Winter
144733      June  Winter

[107502 rows x 25 columns]
```

[121]:
```python
X_train_enc, y_train, X_test_enc, y_test, preprocessor = preprocess_features(
    train_df,
    test_df,
    numeric_features,
    categorical_features + ["Season"],
    drop_features + ["Month"],
)
```

[122]:
```python
X_train_enc.columns
```

[122]:
```
Index(['MinTemp', 'MaxTemp', 'Rainfall', 'Evaporation', 'Sunshine',
       'WindGustSpeed', 'WindSpeed9am', 'WindSpeed3pm', 'Humidity9am',
       'Humidity3pm',
       …
       'x3_W', 'x3_WNW', 'x3_WSW', 'x4_?', 'x4_No', 'x4_Yes', 'x5_Autumn',
       'x5_Fall', 'x5_Summer', 'x5_Winter'],
      dtype='object', length=123)
```

[123]:
```python
coeff_df = score_lr_print_coeff(
    preprocessor, train_df, y_train, test_df, y_test, X_train_enc.columns
)
```

```
Train score: 0.85
Test score: 0.84
```

[124]:
```python
coeff_df.loc[["x5_Fall", "x5_Summer", "x5_Winter", "x5_Autumn"]]
```

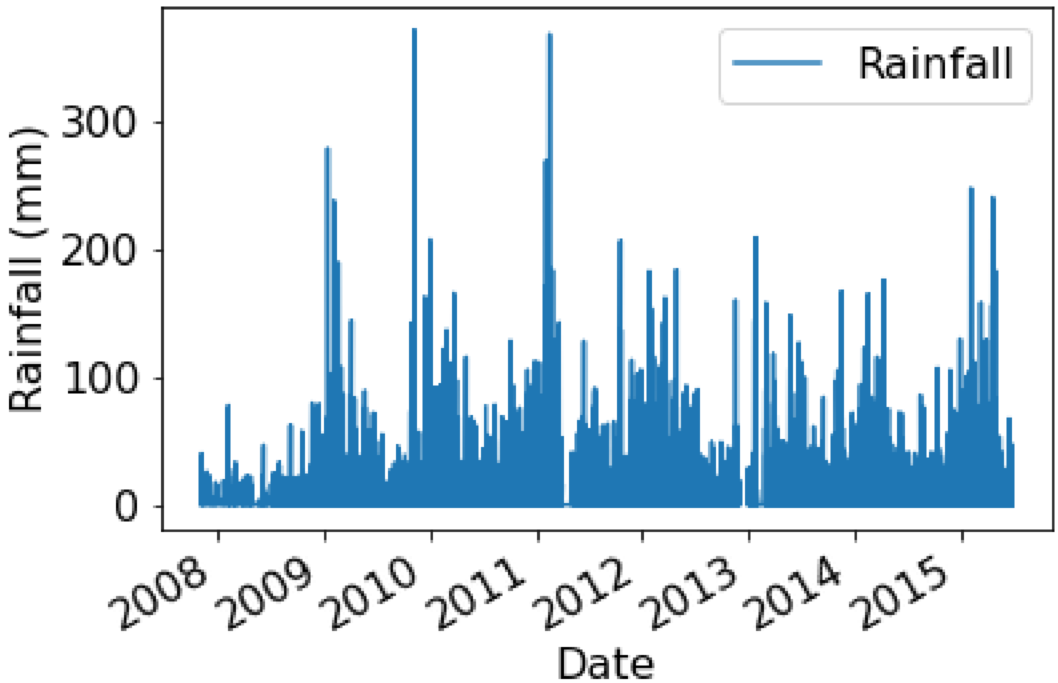
[124]:

	Coef
x5_Fall	0.063903
x5_Summer	-0.225374
x5_Winter	0.105379
x5_Autumn	0.044957

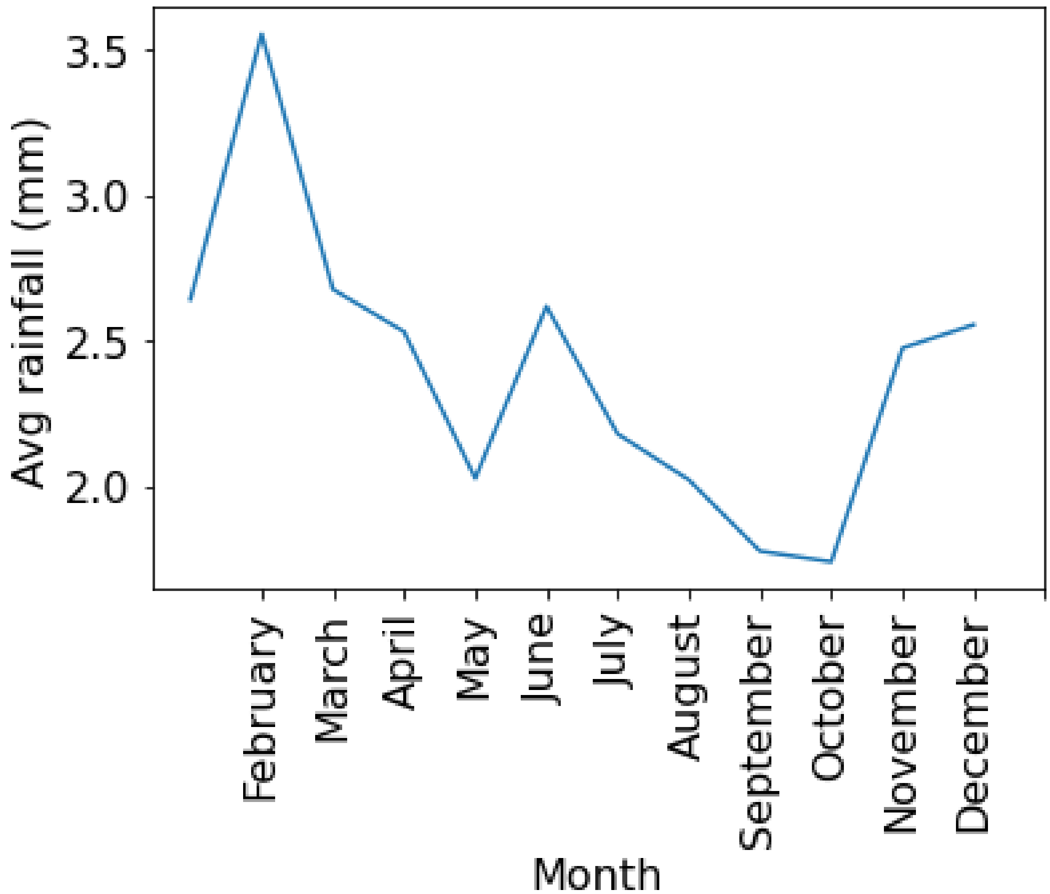
- No improvements in the scores but the coefficients make some sense,
- A negative coefficient for summer and a positive coefficients for winter.

Let's explore Date/Rainfall plots

[125]:
```
train_df.plot(x="Date", y="Rainfall")
plt.ylabel("Rainfall (mm)");
```



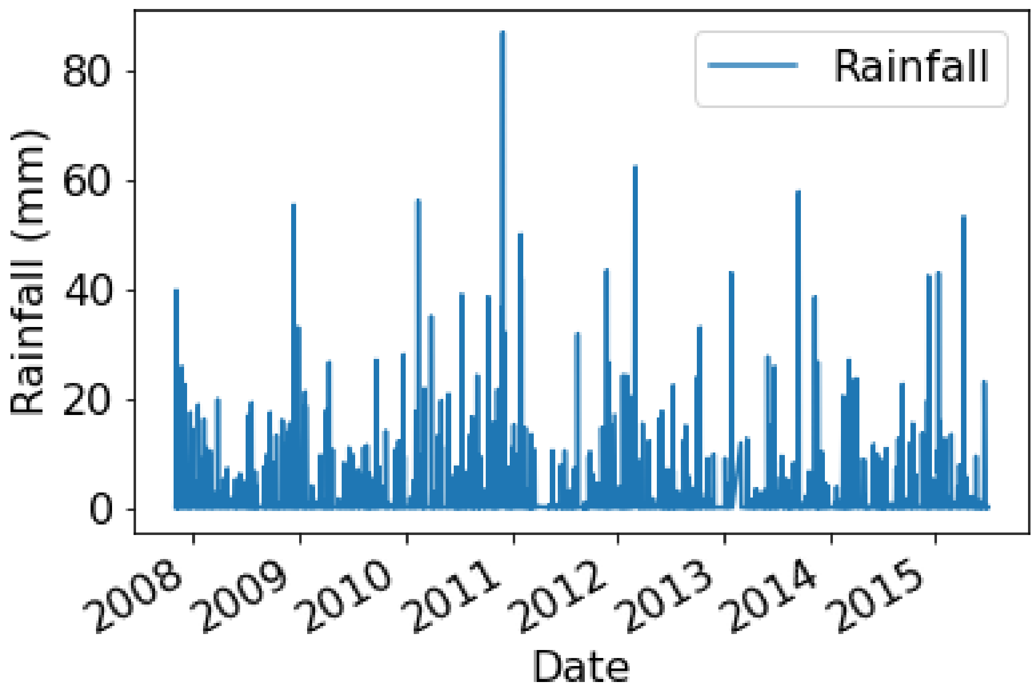
[126]:
```
monthly_avg_rainfall = train_df.groupby("Month")["Rainfall"].mean()
plt.plot(monthly_avg_rainfall)
plt.xticks(np.arange(1, 13).astype(int))
plt.ylabel("Avg rainfall (mm)")
plt.xlabel("Month")
plt.xticks(rotation=90);
```



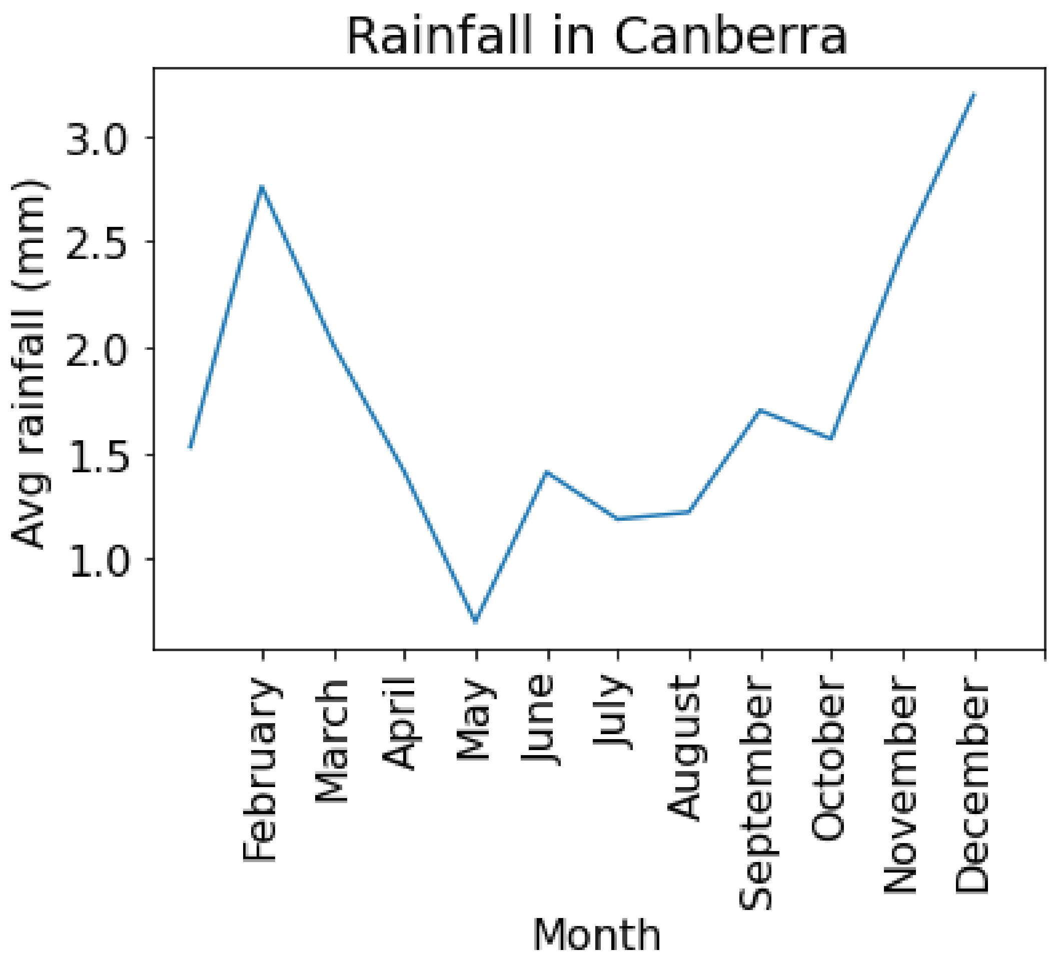
- It's interesting that June rainy but May and August are less so.
- But, Australia is a huge country. Perhaps we should drill down to particular locations:

[127]:
```
train_df_canberra = train_df.query('Location == "Canberra"')
```

[128]:
```
train_df_canberra.plot(x="Date", y="Rainfall")
plt.ylabel("Rainfall (mm)");
```



[129]:
```
plt.plot(train_df_canberra.groupby("Month")["Rainfall"].mean())
plt.xticks(np.arange(1, 13).astype(int))
plt.ylabel("Avg rainfall (mm)")
plt.xlabel("Month")
plt.title("Rainfall in Canberra")
plt.xticks(rotation=90);
```



[130]:
```
train_df_canberra.shape
```

[130]:
```
(2696, 25)
```

- This looks somewhat cleaner but also pretty surprising - why is December so much higher than January?
- Let's find the location with max rainfall

[131]:
```
loc_rainfall = train_df.groupby("Location")["Rainfall"].mean()
loc_rainfall.head()
```

[131]:
```
Location
Adelaide         1.526183
Albany           2.292733
Albury           1.978531
AliceSprings     0.899654
BadgerysCreek    2.282515
```

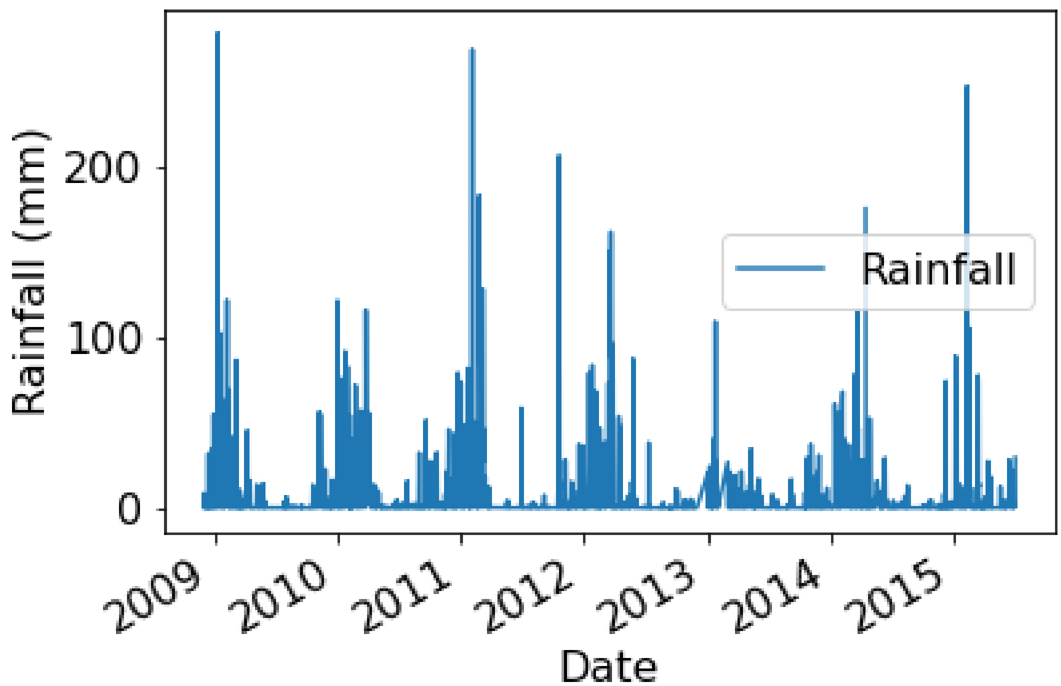
```
Name: Rainfall, dtype: float64
```

[132]:
```
loc_rainfall.idxmax()  # location with max rainfall
```

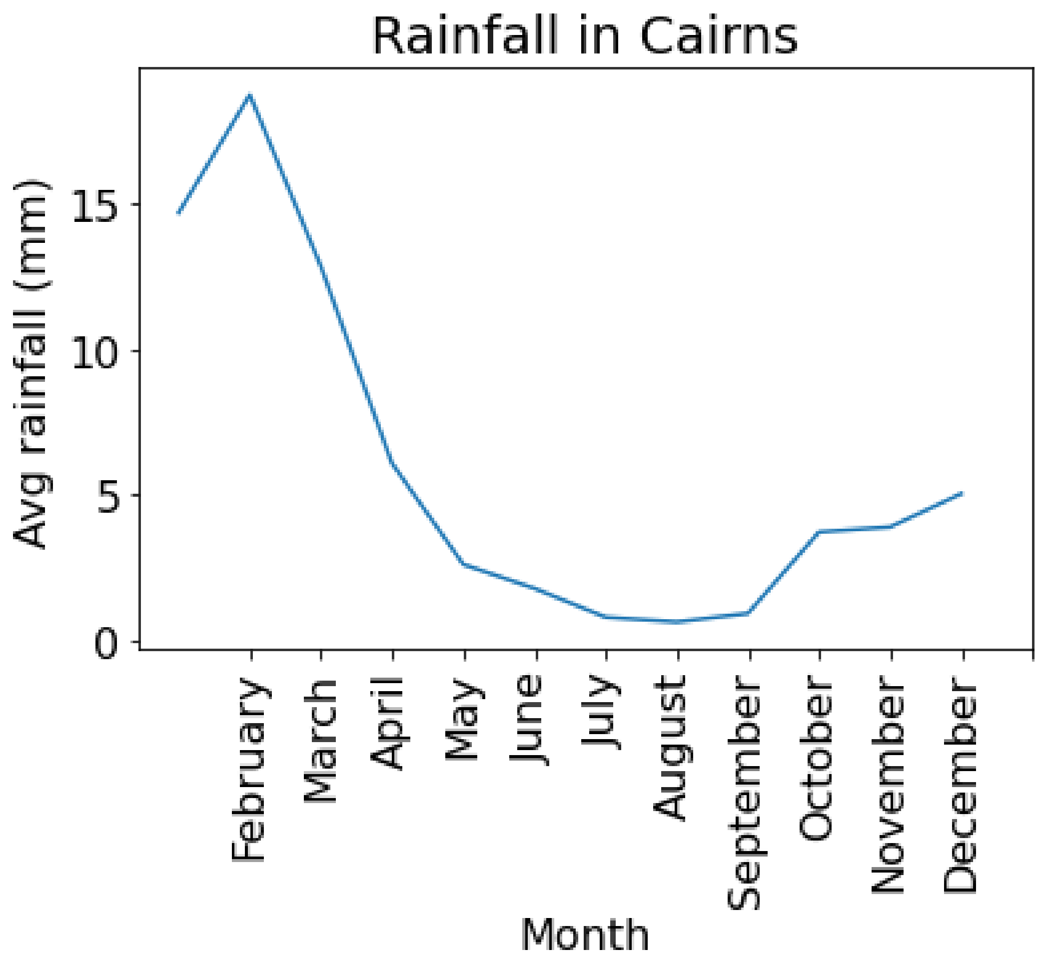
[132]:
```
'Cairns'
```

[133]:
```
train_df_cairns = train_df.query('Location == "Cairns"')
```

[134]:
```
train_df_cairns.plot(x="Date", y="Rainfall")
plt.ylabel("Rainfall (mm)");
```



[135]:
```
plt.plot(train_df_cairns.groupby("Month")["Rainfall"].mean())
plt.xticks(np.arange(1, 13).astype(int))
plt.ylabel("Avg rainfall (mm)")
plt.xlabel("Month")
plt.title("Rainfall in Cairns")
plt.xticks(rotation=90);
```



[136]:
```
train_df_cairns.shape
```

[136]: (2310, 25)

- This looks much cleaner!

1.7 Lag-based features

- In time series data there is **temporal dependence**;
  - observations close in time tend to be correlated.
- Currently we're using features about today to predict tomorrow's rainfall.
- But, what if tomorrow's rainfall is also related to yesterday's features, or the day before?
  - This is called a ***lagged* feature**.
- In time series analysis, we'd look at something called an autocorrelation function (ACF), but we won't go into that here.
- Instead, we can just add those features:

[137]:
```
train_df = rain_df.query("Date <= 20150630")
test_df = rain_df.query("Date >  20150630")
```

[138]:
```
train_df
```

[138]:
```
              Date Location  MinTemp  MaxTemp  Rainfall  Evaporation  Sunshine  \
0       2008-12-01   Albury     13.4     22.9       0.6          NaN       NaN
1       2008-12-02   Albury      7.4     25.1       0.0          NaN       NaN
2       2008-12-03   Albury     12.9     25.7       0.0          NaN       NaN
3       2008-12-04   Albury      9.2     28.0       0.0          NaN       NaN
4       2008-12-05   Albury     17.5     32.3       1.0          NaN       NaN
…              …        …       …        …         …           …
144729 2015-06-26    Uluru      3.8     18.3       0.0          NaN       NaN
144730 2015-06-27    Uluru      2.5     17.1       0.0          NaN       NaN
144731 2015-06-28    Uluru      4.5     19.6       0.0          NaN       NaN
144732 2015-06-29    Uluru      7.6     22.0       0.0          NaN       NaN
144733 2015-06-30    Uluru      6.8     21.1       0.0          NaN       NaN

       WindGustDir  WindGustSpeed WindDir9am  … Humidity9am  Humidity3pm  \
0                W           44.0          W  …        71.0         22.0
1              WNW           44.0        NNW  …        44.0         25.0
2              WSW           46.0          W  …        38.0         30.0
3               NE           24.0         SE  …        45.0         16.0
4                W           41.0        ENE  …        82.0         33.0
…                …              …          …  …           …            …
144729           E           39.0        ESE  …        73.0         37.0
144730           E           41.0        ESE  …        69.0         40.0
144731         ENE           35.0        ESE  …        69.0         39.0
144732         ESE           33.0         SE  …        67.0         37.0
144733         ESE           35.0        ESE  …        81.0         35.0

        Pressure9am  Pressure3pm  Cloud9am  Cloud3pm  Temp9am  Temp3pm  \
0            1007.7       1007.1       8.0       NaN     16.9     21.8
1            1010.6       1007.8       NaN       NaN     17.2     24.3
2            1007.6       1008.7       NaN       2.0     21.0     23.2
3            1017.6       1012.8       NaN       NaN     18.1     26.5
4            1010.8       1006.0       7.0       8.0     17.8     29.7
…                 …            …         …         …        …        …
144729       1031.5       1027.6       NaN       NaN      8.8     17.2
144730       1029.9       1026.0       NaN       NaN      7.0     15.7
144731       1028.7       1025.0       NaN       3.0      8.9     18.0
144732       1027.2       1023.8       6.0       7.0     11.7     21.5
144733       1028.6       1025.2       3.0       NaN     10.6     20.2

        RainToday  RainTomorrow
0              No            No
1              No            No
```

```
2               No          No
3               No          No
4               No          No
…          …          …
144729          No          No
144730          No          No
144731          No          No
144732          No          No
144733          No          No

[107502 rows x 23 columns]
```

- It looks like the dataframe is already sorted by Location and then by date for each Location.
- We could have done this ourselves with:

[139]:
```
# train_df.sort_values(by=["Location", "Date"])
```

But make sure to also sort the targets (i.e. do this before preprocessing).

We can “lag” (or “shift”) a time series in Pandas with the .shift() method.

[140]:
```
train_df = train_df.assign(Rainfall_lag1=train_df["Rainfall"].shift(1))
```

[141]:
```
train_df[["Date", "Location", "Rainfall", "Rainfall_lag1"]].head(10)
```

[141]:
```
         Date Location  Rainfall  Rainfall_lag1
0 2008-12-01   Albury       0.6            NaN
1 2008-12-02   Albury       0.0            0.6
2 2008-12-03   Albury       0.0            0.0
3 2008-12-04   Albury       0.0            0.0
4 2008-12-05   Albury       1.0            0.0
5 2008-12-06   Albury       0.2            1.0
6 2008-12-07   Albury       0.0            0.2
7 2008-12-08   Albury       0.0            0.0
8 2008-12-09   Albury       0.0            0.0
9 2008-12-10   Albury       1.4            0.0
```

- But we have **multiple time series** here and we need to be more careful with this.
- When we switch from one location to another we do not want to take the value from the previous location.

[142]:
```
def create_lag_feature(df, orig_feature, lag):
    """Creates a new df with a new feature that's a lagged version of the␣
↪original, where lag is an int."""
    # note: pandas .shift() kind of does this for you already, but oh well I␣
↪already wrote this code

    new_df = df.copy()
    new_feature_name = "%s_lag%d" % (orig_feature, lag)
```

```
    new_df[new_feature_name] = np.nan
    for location, df_location in new_df.groupby(
        "Location"
    ):  # Each location is its own time series
        new_df.loc[df_location.index[lag:], new_feature_name] = df_location.
↪iloc[:-lag][
            orig_feature
        ].values
    return new_df
```

[143]:
```
train_df = create_lag_feature(train_df, "Rainfall", 1)
```

[144]:
```
train_df[["Date", "Location", "Rainfall", "Rainfall_lag1"]][2285:2295]
```

[144]:
```
           Date      Location  Rainfall  Rainfall_lag1
2309 2015-06-26        Albury       0.2            1.0
2310 2015-06-27        Albury       0.0            0.2
2311 2015-06-28        Albury       0.2            0.0
2312 2015-06-29        Albury       0.0            0.2
2313 2015-06-30        Albury       0.0            0.0
3040 2009-01-01  BadgerysCreek       0.0            NaN
3041 2009-01-02  BadgerysCreek       0.0            0.0
3042 2009-01-03  BadgerysCreek       0.0            0.0
3043 2009-01-04  BadgerysCreek       0.0            0.0
3044 2009-01-05  BadgerysCreek       0.0            0.0
```

Now it looks good!

- Question: is it OK to do this to the test set? Discuss.
- It's fine if you would have this information available in deployment.
- If we're just forecasting the next day, we should.
- Let's include it for now.

[145]:
```
rain_df_modified = create_lag_feature(rain_df, "Rainfall", 1)
train_df = rain_df_modified.query("Date <= 20150630")
test_df = rain_df_modified.query("Date >  20150630")
```

[146]:
```
rain_df_modified
```

[146]:
```
              Date Location  MinTemp  MaxTemp  Rainfall  Evaporation  Sunshine  \
0       2008-12-01   Albury     13.4     22.9       0.6          NaN       NaN
1       2008-12-02   Albury      7.4     25.1       0.0          NaN       NaN
2       2008-12-03   Albury     12.9     25.7       0.0          NaN       NaN
3       2008-12-04   Albury      9.2     28.0       0.0          NaN       NaN
4       2008-12-05   Albury     17.5     32.3       1.0          NaN       NaN
…              …        …        …        …         …           …
145454  2017-06-20    Uluru      3.5     21.8       0.0          NaN       NaN
145455  2017-06-21    Uluru      2.8     23.4       0.0          NaN       NaN
```

```
145456 2017-06-22    Uluru      3.6      25.3       0.0           NaN        NaN
145457 2017-06-23    Uluru      5.4      26.9       0.0           NaN        NaN
145458 2017-06-24    Uluru      7.8      27.0       0.0           NaN        NaN

       WindGustDir  WindGustSpeed WindDir9am  … Humidity3pm  Pressure9am  \
0                W           44.0          W  …        22.0       1007.7
1              WNW           44.0        NNW  …        25.0       1010.6
2              WSW           46.0          W  …        30.0       1007.6
3               NE           24.0         SE  …        16.0       1017.6
4                W           41.0        ENE  …        33.0       1010.8
…              …            …          …  …         …          …
145454           E           31.0        ESE  …        27.0       1024.7
145455           E           31.0         SE  …        24.0       1024.6
145456         NNW           22.0         SE  …        21.0       1023.5
145457           N           37.0         SE  …        24.0       1021.0
145458          SE           28.0        SSE  …        24.0       1019.4

        Pressure3pm  Cloud9am  Cloud3pm  Temp9am  Temp3pm  RainToday  \
0            1007.1       8.0       NaN     16.9     21.8         No
1            1007.8       NaN       NaN     17.2     24.3         No
2            1008.7       NaN       2.0     21.0     23.2         No
3            1012.8       NaN       NaN     18.1     26.5         No
4            1006.0       7.0       8.0     17.8     29.7         No
…             …       …        …       …       …        …
145454       1021.2       NaN       NaN      9.4     20.9         No
145455       1020.3       NaN       NaN     10.1     22.4         No
145456       1019.1       NaN       NaN     10.9     24.5         No
145457       1016.8       NaN       NaN     12.5     26.1         No
145458       1016.5       3.0       2.0     15.1     26.0         No

        RainTomorrow  Rainfall_lag1
0                 No            NaN
1                 No            0.6
2                 No            0.0
3                 No            0.0
4                 No            0.0
…              …            …
145454            No            0.0
145455            No            0.0
145456            No            0.0
145457            No            0.0
145458            No            0.0

[142193 rows x 24 columns]
```

[147]:
```
X_train_enc, y_train, X_test_enc, y_test, preprocessor = preprocess_features(
    train_df,
```

```python
    test_df,
    numeric_features + ["Rainfall_lag1"],
    categorical_features,
    drop_features,
)
```

[148]:
```python
lr_coef = score_lr_print_coeff(
    preprocessor, train_df, y_train, test_df, y_test, X_train_enc.columns
)
# lr_coef
```

```
Train score: 0.85
Test score: 0.84
```

[149]:
```python
lr_coef.loc[["Rainfall", "Rainfall_lag1"]]
```

[149]:

	Coef
Rainfall	0.081068
Rainfall_lag1	0.008305

- Rainfall from today has a positive coefficient.
- Rainfall from yesterday has a positive but a smaller coefficient.
- If we **didn't have rainfall from today** feature, rainfall from **yesterday** feature would have received a **bigger coefficient**.

- We could also create a lagged version of the target.
- In fact, this dataset already has that built in! `RainToday` is the lagged version of the target `RainTomorrow`.
- We could also create lagged version of other features, or more lags

[150]:
```python
rain_df_modified = create_lag_feature(rain_df, "Rainfall", 1)
rain_df_modified = create_lag_feature(rain_df_modified, "Rainfall", 2)
rain_df_modified = create_lag_feature(rain_df_modified, "Rainfall", 3)
rain_df_modified = create_lag_feature(rain_df_modified, "Humidity3pm", 1)
```

[151]:
```python
rain_df_modified[
    [
        "Date",
        "Location",
        "Rainfall",
        "Rainfall_lag1",
        "Rainfall_lag2",
        "Rainfall_lag3",
        "Humidity3pm",
        "Humidity3pm_lag1",
    ]
].head(10)
```

[151]:

```
         Date Location  Rainfall  Rainfall_lag1  Rainfall_lag2  Rainfall_lag3  \
0 2008-12-01   Albury       0.6            NaN            NaN            NaN
1 2008-12-02   Albury       0.0            0.6            NaN            NaN
2 2008-12-03   Albury       0.0            0.0            0.6            NaN
3 2008-12-04   Albury       0.0            0.0            0.0            0.6
4 2008-12-05   Albury       1.0            0.0            0.0            0.0
5 2008-12-06   Albury       0.2            1.0            0.0            0.0
6 2008-12-07   Albury       0.0            0.2            1.0            0.0
7 2008-12-08   Albury       0.0            0.0            0.2            1.0
8 2008-12-09   Albury       0.0            0.0            0.0            0.2
9 2008-12-10   Albury       1.4            0.0            0.0            0.0

   Humidity3pm  Humidity3pm_lag1
0         22.0               NaN
1         25.0              22.0
2         30.0              25.0
3         16.0              30.0
4         33.0              16.0
5         23.0              33.0
6         19.0              23.0
7         19.0              19.0
8          9.0              19.0
9         27.0               9.0
```

Note the pattern of NaN values.

[152]:

```
train_df = rain_df_modified.query("Date <= 20150630")
test_df = rain_df_modified.query("Date >  20150630")
```

[153]:

```
X_train_enc, y_train, X_test_enc, y_test, preprocessor = preprocess_features(
    train_df,
    test_df,
    numeric_features
    + ["Rainfall_lag1", "Rainfall_lag2", "Rainfall_lag3", "Humidity3pm_lag1"],
    categorical_features,
    drop_features,
)
```

[154]:

```
lr_coef = score_lr_print_coeff(
    preprocessor, train_df, y_train, test_df, y_test, X_train_enc.columns
)
```

```
Train score: 0.85
Test score: 0.85
```

[155]:

```
lr_coef.loc[
    [
        "Rainfall",
```

```
        "Rainfall_lag1",
        "Rainfall_lag2",
        "Rainfall_lag3",
        "Humidity3pm",
        "Humidity3pm_lag1",
    ]
]
```

[155]:

```
                       Coef
Rainfall           0.108510
Rainfall_lag1      0.023072
Rainfall_lag2      0.018270
Rainfall_lag3      0.017805
Humidity3pm        1.278646
Humidity3pm_lag1 -0.267520
```

Note the **pattern in the magnitude** of the coefficients.

1.8 Forecasting further into the future

- Let's say we want to predict 7 days into the future instead of one day.
- There are a few main approaches here:

1. Train a separate model for **each number of days**.
   - E.g. one model that predicts RainTomorrow, another model that predicts RainIn2Days, etc. We can build these datasets.
2. Use a **multi-output model** that jointly predicts RainTomorrow, RainIn2Days, etc. However, multi-output models are outside the scope of CPSC 330.
3. Use **one model** and **sequentially predict** using a `for` loop.
   - However, this **requires predicting *all* features** into a model so may not be that useful here.

- To briefly dig into approach 3, this is easier to understand for a univariate (one feature) time series.
- To dig into this we'll look at the Retail Sales of Clothing and Clothing Accessory Stores dataset made available by the Federal Reserve Bank of St. Louis.

[156]:

```
retail_df = pd.read_csv("data/MRTSSM448USN.csv", parse_dates=["DATE"])
retail_df.columns = ["date", "sales"]
```

[157]:

```
retail_df.head()
```

[157]:

	date	sales
0	1992-01-01	6938
1	1992-02-01	7524
2	1992-03-01	8475
3	1992-04-01	9401
4	1992-05-01	9558

[158]:
```
retail_df["date"].min()
```

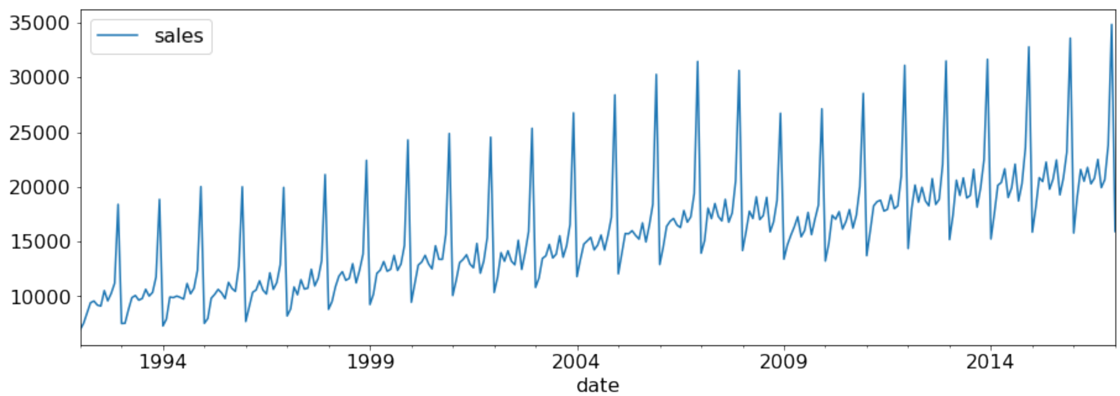
[158]: Timestamp('1992-01-01 00:00:00')

[159]:
```
retail_df["date"].max()
```

[159]: Timestamp('2022-04-01 00:00:00')

[160]:
```
retail_df_train = retail_df.query("date <= 20170101")
retail_df_test = retail_df.query("date >  20170101")
```

[161]:
```
retail_df_train.plot(x="date", y="sales", figsize=(15, 5));
```



We can create a dataset using purely lag features.

[162]:
```
def lag_df(df, lag, cols):
    return df.assign(
        **{f"{col}-{n}": df[col].shift(n) for n in range(1, lag + 1) for col in␣
 ↪cols}
    )
```

[163]:
```
retail_lag_5 = lag_df(retail_df, 5, ["sales"])
retail_train_5 = retail_lag_5.query("date <= 20170101")
retail_test_5 = retail_lag_5.query("date >  20170101")
retail_train_5
```

[163]:

	date	sales	sales-1	sales-2	sales-3	sales-4	sales-5
0	1992-01-01	6938	NaN	NaN	NaN	NaN	NaN
1	1992-02-01	7524	6938.0	NaN	NaN	NaN	NaN
2	1992-03-01	8475	7524.0	6938.0	NaN	NaN	NaN
3	1992-04-01	9401	8475.0	7524.0	6938.0	NaN	NaN
4	1992-05-01	9558	9401.0	8475.0	7524.0	6938.0	NaN
..	…	…	…	…	…	…	…

```
296 2016-09-01  19928  22505.0  20782.0  20274.0  21774.0  20514.0
297 2016-10-01  20650  19928.0  22505.0  20782.0  20274.0  21774.0
298 2016-11-01  23826  20650.0  19928.0  22505.0  20782.0  20274.0
299 2016-12-01  34847  23826.0  20650.0  19928.0  22505.0  20782.0
300 2017-01-01  15921  34847.0  23826.0  20650.0  19928.0  22505.0

[301 rows x 7 columns]
```

- Now, if we drop the "date" column we have
  - a target ("sales") and
  - 5 features (the previous 5 days of sales).
- We need to impute/drop the missing values and then we can fit a model to this. I will just drop for convenience:

[164]:
```
retail_train_5 = retail_train_5[5:].drop(columns=["date"])
retail_train_5
```

[164]:
```
     sales  sales-1  sales-2  sales-3  sales-4  sales-5
5     9182   9558.0   9401.0   8475.0   7524.0   6938.0
6     9103   9182.0   9558.0   9401.0   8475.0   7524.0
7    10513   9103.0   9182.0   9558.0   9401.0   8475.0
8     9573  10513.0   9103.0   9182.0   9558.0   9401.0
9    10254   9573.0  10513.0   9103.0   9182.0   9558.0
..     …        …        …        …        …        …
296  19928  22505.0  20782.0  20274.0  21774.0  20514.0
297  20650  19928.0  22505.0  20782.0  20274.0  21774.0
298  23826  20650.0  19928.0  22505.0  20782.0  20274.0
299  34847  23826.0  20650.0  19928.0  22505.0  20782.0
300  15921  34847.0  23826.0  20650.0  19928.0  22505.0

[296 rows x 6 columns]
```

[165]:
```
retail_train_5_X = retail_train_5.drop(columns=["sales"])
retail_train_5_y = retail_train_5["sales"]
```

[166]:
```
from sklearn.ensemble import RandomForestRegressor
```

[167]:
```
retail_model = RandomForestRegressor()
retail_model.fit(retail_train_5_X, retail_train_5_y);
```

Given this, we can now predict the sales

[168]:
```
preds = retail_model.predict(retail_test_5.drop(columns=["date", "sales"]))
preds
```

[168]:
```
array([18616.48, 20415.87, 21694.89, 22467.23, 21114.88, 22269.33,
       22125.53, 22845.07, 20989.31, 22820.98, 30887.77, 15881.83,
       18627.31, 21130.62, 22240.74, 21868.71, 28765.65, 21034.67,
```

```
       22093.24, 29779.64, 21708.63, 21985.46, 21486.28, 15932.  ,
       18772.01, 20404.79, 22391.84, 22212.39, 28360.13, 22081.68,
       21925.26, 30677.25, 21577.89, 21948.33, 27823.64, 15941.43,
       19031.6 , 21439.78, 13157.29, 11188.87, 10875.16, 14550.56,
       11912.13,  9623.18, 18518.62, 20691.91, 23083.33, 14687.02,
       17664.73, 18970.94, 31352.74, 29340.92, 15990.5 , 28299.48,
       26286.69, 28365.17, 23821.67, 29063.58, 16112.38, 16522.07,
       20382.02, 22711.27, 19152.86])
```

[169]:

```
retail_test_5_preds = retail_test_5.assign(predicted_sales=preds)
retail_test_5_preds.head()
```

[169]:

```
           date  sales  sales-1  sales-2  sales-3  sales-4  sales-5  \
301  2017-02-01  18036  15921.0  34847.0  23826.0  20650.0  19928.0
302  2017-03-01  21348  18036.0  15921.0  34847.0  23826.0  20650.0
303  2017-04-01  21154  21348.0  18036.0  15921.0  34847.0  23826.0
304  2017-05-01  21954  21154.0  21348.0  18036.0  15921.0  34847.0
305  2017-06-01  20623  21954.0  21154.0  21348.0  18036.0  15921.0

     predicted_sales
301         18616.48
302         20415.87
303         21694.89
304         22467.23
305         21114.88
```

- Ok, that is fine, but what if we want to predict 7 days in the future?
- Well, we would not have access to our features!! We don't yet know the previous day's sales, or 2 days prior!
- So we can use "Approach 3" mentioned earlier:
  - predict these values and then pretend they are true!
- For simplicity, say today is Monday

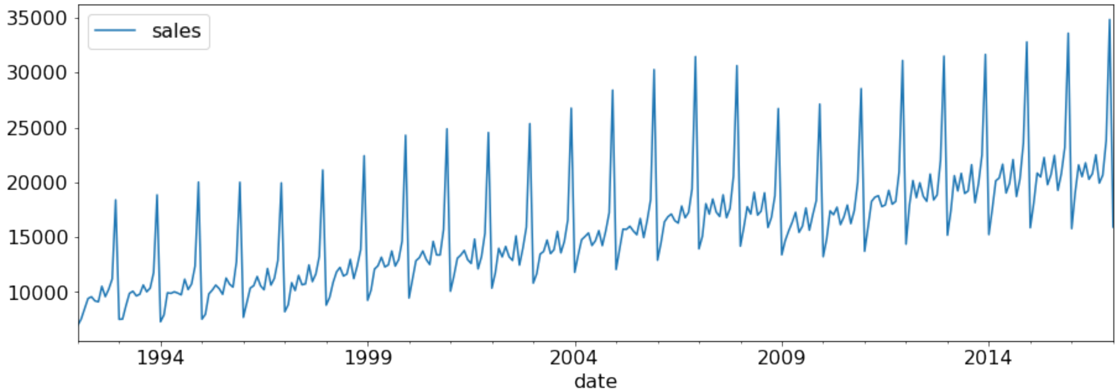
1. **Predict** Tuesday's sales
2. Then, to predict for Wednesday, we need to know Tuesday's sales. Use our *prediction* for Tuesday as the truth.
3. Then, to predict for Thursday, we need to know Tue and Wed sales. Use our predictions.
4. Etc etc.

1.9 Trends

- There are some important concepts in time series that rely on having a continuous target (like we do in the retail sales example above).
- Part of that is the idea of seasonality and trends.
- These are mostly taken care of by our feature engineering of the data variable, but there's something important left to discuss.

[170]:

```
retail_df_train.plot(x="date", y="sales", figsize=(15, 5));
```



- It looks like there's a **trend** here - the sales are going up over time.

Let's say we encoded the date as a feature in days like this:

[171]:
```
retail_train_5_date = retail_lag_5.query("date <= 20170101")
first_day_retail = retail_train_5_date["date"].min()

retail_train_5_date = retail_train_5_date.assign(
    Days_since=retail_train_5_date["date"].apply(lambda x: (x -
 ↪first_day_retail).days)
)
retail_train_5_date.head(10)
```

[171]:

	date	sales	sales-1	sales-2	sales-3	sales-4	sales-5	Days_since
0	1992-01-01	6938	NaN	NaN	NaN	NaN	NaN	0
1	1992-02-01	7524	6938.0	NaN	NaN	NaN	NaN	31
2	1992-03-01	8475	7524.0	6938.0	NaN	NaN	NaN	60
3	1992-04-01	9401	8475.0	7524.0	6938.0	NaN	NaN	91
4	1992-05-01	9558	9401.0	8475.0	7524.0	6938.0	NaN	121
5	1992-06-01	9182	9558.0	9401.0	8475.0	7524.0	6938.0	152
6	1992-07-01	9103	9182.0	9558.0	9401.0	8475.0	7524.0	182
7	1992-08-01	10513	9103.0	9182.0	9558.0	9401.0	8475.0	213
8	1992-09-01	9573	10513.0	9103.0	9182.0	9558.0	9401.0	244
9	1992-10-01	10254	9573.0	10513.0	9103.0	9182.0	9558.0	274

- Now, let's say we use all these features (the lagged version of the target and also `Days_since`.
- If we use **linear regression** we'll learn a coefficient for `Days_since`.
  - If that coefficient is positive, it predicts unlimited growth forever. That may not be what you want? It depends.
- If we use a **random forest**, we'll just be doing splits from the training set, e.g. "if `Days_since` > 9100 then do this".
  - There will be no splits for later time points because there is no training data there.
  - Thus **tree-based models cannot model trends**.
  - This is really important to know!!

- Often, we **model the trend separately** and
  - use the **random forest** to model a **de-trended** time series.

1.10 What did we not cover?

- A huge amount!

1.10.1 Traditional time series approaches

- Time series analysis is a huge field of its own
- Traditional approaches include the ARIMA model and its various components/extensions.
- In Python, the statsmodels package is the place to go for this sort of thing.
  - For example, statsmodels.tsa.arima_model.ARIMA.
- These **approaches can forecast**, and
  - they are also very good for understanding the **temporal relationships** in your data.
- We will take a different route in this course, and **stick to our supervised learning** tools.

1.10.2 Deep learning

- Recently, deep learning has been very successful too.
- In particular, recurrent neural networks (RNNs).
  - These are not covered in CPSC 340, but I believe they are in 540 (soon to be renamed 440).
  - LSTMs especially have shown a lot of promise in this type of task.
  - Here is a blog post about LSTMs.

1.10.3 Types of problems involving time series

- A **single label** associated with an entire time series.
  - We had that with images earlier on, you could have the same for a time series.
  - E.g., for fraud detection, labelling each transaction as fraud/normal vs. labelling a person as bad/good based on their entire history.
  - There are various approaches that can be used for this type of problem, including CNNs (Lecture 14), LSTMs, and non deep learning methods.
- **Inference** problems.
  - What are the **patterns in this time series**?
  - How **many lags** are associated with the current value?
- Etc.

Unequally spaced time points

- We assumed we have a **measurement each day**.
- For example, when creating lag features we used consecutive rows in the DataFrame.
- But, in fact **some days were missing** in this dataset.
- More generally, what if the measurements are at arbitrary times, not equally spaced?
  - Some of our approaches would **still work**, like encoding the month / looking at seasonality.
  - Some of our approaches would **not make sense**, like the lags.
  - Perhaps the measurements could be binned into **equally spaced bins**, or something.
  - This is more of a hassle.

Other software package

- One good one to know about is Prophet.

Feature engineering

- Often, a useful approach is to just ***engineer your own features***.
  - E.g., max expenditure, min expenditure, max-min, avg time gap between transactions, variance of time gap between transactions, etc etc.
  - We could do that here as well, or in any problem.