```
# data wrangling code for Monterey Airport Weather Almanacs
# Jeff Trevino, 2019

from datetime import datetime

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# pandas options
pd.set_option('display.max_columns', 125) # csv contains 124 columns
pd.set_option('display.max_rows', 4000) # display more rows

data = pd.read_csv('montereyClimateData.csv')

df = data
columns = ['DATE',
           'HourlySkyConditions',
           'HourlyVisibility',
           'HourlyDryBulbTemperature',
           'HourlyWindSpeed',
           'DailyMaximumDryBulbTemperature',
           'DailyMinimumDryBulbTemperature',
           'DailyPeakWindSpeed',
           'DailyPrecipitation',
           'HourlyRelativeHumidity'
          ]
df = df.loc[:, columns]

def date_val_to_datetime(to_parse):
    to_format = to_parse.split('T')
    return datetime.strptime(to_format[0] + ' ' + to_format[1], '%Y-%m-%d
%H:%M:%S')

df['datetime'] = df.loc[:,'DATE'].apply(date_val_to_datetime)

df = df.set_index(['datetime'])

cols = ['HourlyVisibility', # columns to convert
 'HourlyDryBulbTemperature',
 'HourlyWindSpeed',
```

```
 'DailyMaximumDryBulbTemperature',
 'DailyMinimumDryBulbTemperature',
 'DailyPeakWindSpeed',
 'DailyPrecipitation',
 'HourlyRelativeHumidity',
       ]

# convert columns by applying to_numeric with error coersion
df.loc[:, cols] = df.loc[:, cols].apply(pd.to_numeric, errors='coerce')

# check for desired result
for c in cols:
    assert df.loc[:, c].dtypes == np.float64
    assert len(df.loc[df[c].astype(str).str[-1].isin(('*', 's'))]) == 0
    # no values have the "suspect" suffix anymore


df.loc[:,['DailyMaximumDryBulbTemperature',
'DailyMinimumDryBulbTemperature', 'DailyPeakWindSpeed',
'DailyPrecipitation']] = df.loc[:,['DailyMaximumDryBulbTemperature',
'DailyMinimumDryBulbTemperature', 'DailyPeakWindSpeed',
'DailyPrecipitation', 'HourlyRelativeHumidity']].bfill()

# column value is a string of a list of codes, 'BKN:07 15 OVC:08 20'
# desired output is a list of tuples, [('BKN', 7, 15), ('OVC', 8, 20)]
# clear days lack a second integer, i.e., 'CLR:00', appending 0 in place
of missing value
from collections import namedtuple

SkyCondition = namedtuple('SkyCondition', 'obscuration,
vertical_distance') # these will be the dict's values

def list_of_lists_by_n(the_list, n):
    """Yields the next n elements of a list as a sublist"""
    for i in range(0, len(the_list), n):
        yield the_list[i:i + n]

def from_many_to_two(the_string):
    split_at_spaces = the_string.split(' ')
    return list(list_of_lists_by_n(split_at_spaces, 2))

def from_two_to_three(list_of_lists):
```

```
78      """
79      input: ['CAPS:02', '35']
80      output: {'CAPS':, SkyCondition(obscuration=02, vertical_distance=35)}
81      """
82      output = []
83      for two_element_list in list_of_lists:
84          first_element = two_element_list[0]
85          if 2 >= len(first_element):
86              return {} # for single trailing ints
87          first_element_split = first_element.split(":")
88          if 2 > len(two_element_list):
89              two_element_list.append(0) # catch CLR days missing following
… 00
90          condition = SkyCondition(int(first_element_split[1]),
… int(two_element_list[1]))
91          output.append({first_element_split[0]: condition})
92      return output
93
94  def condition_string_to_namedtuple_dict(value):
95      """
96      Converts string containing several of the following to a list of
… dictionaries as follows:
97      input: "CAPS:03 34"
98      output: {'CAPS':, SkyCondition(obscuration=3, vertical_distance=34)}
99      """
100     if isinstance(value, float): # the only floats are np.nan, which is a
… float...with a str repr
101         return [] # replace NaNs as an empty list
102     the_string = value
103     list_of_twos = from_many_to_two(the_string)
104     return from_two_to_three(list_of_twos)
105
106 df['HourlySkyConditions'] =
… df['HourlySkyConditions'].apply(condition_string_to_namedtuple_dict)
107
108 def calculate_average_obscuration(sky_conditions_for_hour):
109     """Calculates the mean obscuration for each hour in the dataset"""
110     if not sky_conditions_for_hour:
111         return np.nan
112     else:
113         obscurations = [[y.obscuration for x, y in d.items()] for d in
… sky_conditions_for_hour]
```

```
114         obscuration_mean = sum([x[0] for x in obscurations]) /
…   len(obscurations) # calculate mean obscuration
115         return obscuration_mean
116
117
118 df['averageObscuration'] =
…   df['HourlySkyConditions'].apply(calculate_average_obscuration)
119
120 # impute the column mean for all remaining nan values in all numeric
…   columns
121 x = df
122 numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
123 numeric_cols = x.select_dtypes(include=numerics)
124
125 df.fillna({x:np.mean(df[x]) for x in numeric_cols.columns}, inplace=True)
126
127 # write out cleaned dataframe to csv
128 df.to_csv('cleaned_df.csv')
129
```

```
# exploratory visualization code for Monterey Airport Weather Almanacs
# Jeff Trevino, 2019
from datetime import date

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv('cleaned_df.csv', parse_dates=['datetime'],
index_col=['datetime'])

sns.set()

hourly_obscuration =
pd.DataFrame(df.groupby(df.index.hour).averageObscuration.mean())
hourly_obscuration = hourly_obscuration.reset_index()
hourly_obscuration.columns = ['hour of the day', 'mean obscuration']
plt.figure(figsize=(16, 6))
sns.barplot(y='hour of the day', x='mean obscuration', color='grey',
orient='h', data=hourly_obscuration).set_title('The Sky Is More Likely to
Be Clear Between 10 AM and 4 PM in Monterey')
plt.show()


f, axes = plt.subplots(24,1)
f.set_size_inches(32,128)
axes = axes.flatten()

def add_subplot(master_frame, index):
    hour_frame = hours.loc[index]
    hour_frame = hour_frame.reset_index()
    v = sns.violinplot(y='averageObscuration', x='month', data=hour_frame,
ax=axes[index])
    v.set_xlabel("month", fontsize=30)
    v.set_ylabel("obscuration", fontsize=30)
    title_string = str(index) + " o'clock"
    v.set_title(title_string, fontsize=30)

hours = df
hours = pd.DataFrame(df.groupby([df.index.hour, df.index.day,
df.index.month]).averageObscuration.mean())
```

```
hours.index = hours.index.set_names(['hour', 'day', 'month'])
for x in range(24):
    add_subplot(hours, x)



left   =  0.125  # the left side of the subplots of the figure
right  =  0.9    # the right side of the subplots of the figure
bottom =  0.1    # the bottom of the subplots of the figure
top    =  0.9    # the top of the subplots of the figure
wspace =  .5     # the amount of width reserved for blank space between subplots
hspace =  1.1    # the amount of height reserved for white space between subplots

# This function actually adjusts the sub plots using the above paramters
plt.subplots_adjust(
    left    =  left,
    bottom  =  bottom,
    right   =  right,
    top     =  top,
    wspace  =  wspace,
    hspace  =  hspace
)
plt.savefig('figures/hourlyAverageObscurationOverYear.png')
plt.show()

plt.figure(figsize=(12, 6))
sns.set_style('darkgrid')

obscuration = pd.DataFrame(df['averageObscuration'].groupby([df.index.date]).mean().dropna()) # ten years of individual dates
obscuration.name = 'There are more cloudy days than sunny days in the last decade.'

plt.xlim(0, 9)
sns.distplot(obscuration,bins=8, kde=False, norm_hist=False)

plt.xlabel('obscuration rating')
plt.ylabel('days')
plt.show()
```

```
74
75 plt.figure(figsize=(12, 6))
76
77 obscuration =
… pd.DataFrame(df['averageObscuration'].groupby([df.index.date]).mean().
… dropna()) # ten years of individual dates
78 obscuration.name = 'There are more cloudy days than sunny days in the last
… decade.'
79 sns.distplot(obscuration,bins=8, kde=False, label='3,400 days throughout
… the decade')
80
81 obscuration_year_averaged_across_decade =
… pd.DataFrame(df['averageObscuration'].groupby([df.index.month,
… df.index.day]).mean())
82 obscuration_year_averaged_across_decade.name = 'Averaging Obscuration
… Across the Ten Years Compresses the Distribution'
83 sns.distplot(obscuration_year_averaged_across_decade,bins=8, kde=False,
… label='365 calendar days averaged across ten years')
84
85
86 # plt.ylim(0, 500)
87 plt.xlim(0, 9)
88 plt.legend()
89 plt.xlabel('obscuration')
90 plt.ylabel('days')
91 plt.show()
92 # plt.savefig('figures/dailyMeanForDatesAcrossDecade.png') # uncomment to
… write out figure
93
94 plt.figure(figsize=(12, 6))
95 obscuration_year_averaged_across_decade =
… pd.DataFrame(df['averageObscuration'].groupby([df.index.month,
… df.index.day]).mean())
96 obscuration_year_averaged_across_decade.name = 'Averaging Obscuration
… Across the Ten Years Compresses the Distribution'
97 sns.distplot(obscuration_year_averaged_across_decade,bins=8,
… color='orange', kde=False, label='365 calendar days averaged across ten
… years')
98
99
100 # plt.ylim(0, 500)
101 plt.xlim(0, 9)
```

```
plt.legend()
plt.xlabel('obscuration')
plt.ylabel('calendar days')
plt.show()
# plt.savefig('figures/dailyMeanForDatesAcrossDecade.png') # uncomment to write out figure

by_date = df[(df.index.hour >= 10) & (df.index.hour <= 16)] # get 10 AM to 4 PM
by_date = df.groupby([df.index.month, df.index.day]).averageObscuration.mean() # average by calendar day across decade
by_date = by_date.sort_values()
by_date = by_date[by_date <= 3.5] # 3.5 is a conservative cut-off for a clear day: there are at worst "scattered clouds"
print(str(len(by_date)) + " days have had a decade average obscuration rating of under 3.5.")

by_date = df[(df.index.hour >= 10) & (df.index.hour <= 16)] # get 10 AM to 4 PM
by_date = df.groupby([df.index.month, df.index.day]).averageObscuration.mean()
by_date = by_date.sort_values()
by_date = by_date[by_date <= 3.5] # 3.5 is a conservative cut-off for a clear day: there are at worst "scattered clouds"
by_date = pd.DataFrame(by_date)
by_date.index = by_date.index.rename(["month", "day"])
# by_date = by_date.unstack(level=0)
by_date
by_date.groupby('month').count().rename(columns={'averageObscuration':'number of clearish days'}).plot(kind='bar', title='A Third of January and November Are Clearish Between 10 AM and 4 PM')
plt.show()

by_date = df[(df.index.hour >= 10) & (df.index.hour <= 16)] # get 10 AM to 4 PM
by_date = df.groupby([df.index.month, df.index.day]).averageObscuration.mean()
by_date = by_date.sort_values()
by_date = by_date[by_date <= 3.5] # 3.5 is a conservative cut-off for a clear day: there are at worst "scattered clouds"
by_date = pd.DataFrame(by_date)
```

```
by_date.index = by_date.index.rename(["month", "day"])

# set up subplots
f, axes = plt.subplots(2,3, sharey='row')
f.set_size_inches(12,12)
axes = axes.flatten()

# set up title lookup
month_dict = {1: 'January', 2: 'February', 3: 'March', 10: 'October', 11:
'November', 12: 'December'}

# plot a month
def plot_month(frame, month_index, plot_index):
    """Plots the decade mean obscuration for the index month's clearish
days"""
    frame = frame.reset_index()
    frame = frame[frame['month'] == month_index]
    frame = frame.set_index('day')
    frame = frame.sort_index()
    b = sns.barplot(data=frame, x=frame.index, color='skyblue',
y='averageObscuration', ax=axes[plot_index])
    b.set_xlabel('day of the month')
    b.set_ylabel('decade mean obscuration')
    b.set_title(month_dict[month_index])

for plot, month in enumerate(set(by_date.index.get_level_values(0))):
    plot_month(by_date, month, plot)

plt.suptitle("36 Calendar Days Have A Decade Mean Obscuration of Less Than
3.5 in Monterey")
plt.savefig('figures/clearishDaysByMonth.png')
plt.show()

plt.figure(figsize=(12, 6))

x = df
max_temp =
x.groupby(df.index.dayofyear)['DailyMaximumDryBulbTemperature'].mean().
rolling(14).mean().plot(label='max')
min_temp =
x.groupby(df.index.dayofyear)['DailyMinimumDryBulbTemperature'].mean().
rolling(14).mean().plot(label='min')
```

```
164
165 x['mean_temp'] = (df['DailyMaximumDryBulbTemperature'] +
  … df['DailyMinimumDryBulbTemperature'])/2
166 mean_temp =
  … x.groupby(df.index.dayofyear)['mean_temp'].mean().rolling(14).mean().plot(
  … label='mean')
167
168 # plot
169 plt.legend()
170 plt.title('Moving Two-Week Average Annual Temperature')
171 plt.xlabel('Day of Year')
172 plt.ylabel('Degrees (F)')
173 plt.show()
174
175 plt.figure(figsize=(12, 6))
176
177 x = df
178 bool_index = (x.index.year >= 2010) & (x.index.year <= 2018)
179 x = x[bool_index]
180 x['mean_temp'] = (x['DailyMaximumDryBulbTemperature'] +
  … x['DailyMinimumDryBulbTemperature'])/2
181
182 mean_temp = x.groupby([x.index.year,
  … x.index.dayofyear])['mean_temp'].mean().rolling(14).mean()
183
184 mean_temp = mean_temp.unstack(level=0)
185
186 # plot
187 for col in mean_temp.columns:
188     plt.plot(mean_temp[col], label=str(col))
189 plt.legend()
190 plt.title('Moving Two-Week Average of Calendar Day Temperature Exhibits
  … Annual Seasonality')
191 plt.xlabel('Day of Year')
192 plt.ylabel('Degrees (F)')
193 plt.show()
194
195 plt.figure(figsize=(12, 12))
196
197 # processing setup
198 x = df
199 x.reset_index()
```

```
200
201 # wrangle: year columns, day of year index, daily high temperature values
202 # mean does nothing here: all entries have same max value
203 x = x.groupby([x.index.month,
… x.index.date]).DailyMaximumDryBulbTemperature.last() # all hourly entries
… have same value
204 x.index = x.index.rename(['month', 'date'])
205 x = x.unstack(level=0)
206 x.head(100)
207
208
209 # # plot
210 b = sns.boxplot(data=x)
211 b.set_title('Summer Month Daily Max Temperatures (F) Have Narrow
… Interquartile Ranges and Few Outliers')
212 b.set_ylabel('Temperature (F)')
213 plt.show()
214
215 plt.figure(figsize=(12, 12))
216
217 # processing setup
218 x = df
219 x.reset_index()
220
221 # wrangle: year columns, day of year index, daily high temperature values
222 x = x.groupby([x.index.month,
… x.index.date]).DailyMinimumDryBulbTemperature.first()
223 x.index = x.index.rename(['month', 'date'])
224 x = x.unstack(level=0)
225 x.head(100)
226
227
228 # # plot
229 sns.boxplot(data=x).set_title('Summer Months Have a Narrower Range of
… Minimum Temperatures')
230 plt.show()
231
232 # processing setup
233 x = df
234 x.reset_index()
235 x.head()
236
```

```
237 # wrangle: create average obscuration and max temperature columns
238 x = pd.DataFrame(x.groupby([x.index.date,
… x['DailyMaximumDryBulbTemperature']]).averageObscuration.mean())
239 x.index = x.index.rename(['date', 'max temp (F)'])
240 x = x.reset_index()
241 x = x.set_index(['date'])
242 x.columns = ['max temp (F)', 'mean obscuration']
243 x.head()
244
245 # plot
246 hexplot = sns.jointplot(x='max temp (F)', y='mean obscuration', height=10,
… data=x, kind='hex')
247 plt.show()
248
249 # processing setup
250 x = df
251 x.reset_index()
252 x.head()
253
254 # wrangle: create average obscuration and min temperature columns
255 x = pd.DataFrame(x.groupby([x.index.date,
… x['DailyMinimumDryBulbTemperature']]).averageObscuration.mean())
256 x.index = x.index.rename(['date', 'max temp (F)'])
257 x = x.reset_index()
258 x = x.set_index(['date'])
259 x.columns = ['min temp (F)', 'mean obscuration']
260 x.head()
261
262 # plot
263 hexplot = sns.jointplot(x='min temp (F)', y='mean obscuration', height=10,
… data=x, kind='hex')
264 plt.show()
265
266 # processing setup
267 x = df
268 x.reset_index()
269 x.head()
270
271 # wrangle: rename wind speed and precipitation columns
272 x = x.rename(columns={'DailyPrecipitation': "Daily Rain (in)",
… 'DailyPeakWindSpeed': 'Daily Max Wind Speed (m/s)'})
273
```

```
# plot
sns.set_style('white')
j = sns.jointplot(x='Daily Rain (in)', y='Daily Max Wind Speed (m/s)', height=10, data=x, kind='reg')
plt.show()

x = df[(df.index.year >= 2010) & (df.index.year < 2019)] # choose 2010 through 2018, because 2009 and 2019 are missing some dates
x = x.groupby([x.index.year, x.index.date]).DailyPrecipitation.sum()
x.index = x.index.rename(['year', 'date'])
x = x[x == 0]
x = x.reset_index()
x = x.groupby('year').count().rename(columns={'DailyPrecipitation':'Days Without Rain'}).drop(columns=['date'])
x

# average rainfall
x = df
x = x.groupby([x.index.year, x.index.date])['DailyPrecipitation'].first()
x.index = x.index.rename(['year', 'date'])
x = x.groupby(['year']).sum().mean()
x

x = df
x = x.groupby([x.index.year, x.index.date])['DailyPrecipitation'].first()
x.index = x.index.rename(['Year', 'Date'])
x = pd.DataFrame(x.groupby(['Year']).sum())
x.columns = ['Annual Precipitation in Inches']
x = x.reset_index()

# plot
sns.set_style('darkgrid')
sns.barplot(x='Year', y='Annual Precipitation in Inches', color='skyblue', data=x)
plt.show()

# daily precipitation for 36 clearest days on calendar
plt.figure(figsize=(12, 6))

by_date = df[(df.index.hour >= 10) & (df.index.hour <= 16)] # get 10 AM to 4 PM (see definition of 'clearish' above)

```

```
311 # mean the average daily obscuration and keep the first value for daily
… precipitaton for each date throughout decade
312 by_date = df.groupby([df.index.date]).agg({'DailyPrecipitation': 'first',
… 'averageObscuration': 'mean'})
313
314 # further average both obscuration and daily rainfall by calendar day
315 by_date = df.groupby([df.index.month,
… df.index.day]).agg({'DailyPrecipitation': 'mean', 'averageObscuration':
… 'mean'})
316 by_date.index = by_date.index.rename(['month', 'day'])
317 by_date.columns = ['mean rain (in)', 'mean obscuration']
318 by_date = by_date.reset_index()
319
320 # filter out clearish days
321 by_date = by_date[by_date['mean obscuration'] <= 3.5] # 3.5 is a
… conservative cut-off for a clear day: there are at worst "scattered
… clouds"
322
323 # sort by ascending rainfall
324 by_date = by_date.sort_values(by='mean rain (in)')
325
326 # add a date column to serve as the index
327 by_date['date'] = by_date.apply(lambda x: date(year=1,
… month=int(x['month']), day=int(x['day'])), axis=1)
328 by_date
329
330
331 # plot
332 b = sns.barplot(x='date', y='mean rain (in)', color='skyblue',
… data=by_date)
333 b.set_title('The 36 Clearish Days Ordered by Decade Average Daily
… Rainfall')
334 plt.xticks(rotation=90)
335 plt.show()
336
337 plt.figure(figsize=(12, 6))
338
339 by_date = df[(df.index.hour >= 10) & (df.index.hour <= 16)] # get 10 AM to
… 4 PM (see definition of 'clearish' above)
340
341 # mean the average daily obscuration and keep the first value for daily
… precipitaton for each date throughout decade
```

```
342 by_date =
… df.groupby([df.index.date]).agg({'DailyMaximumDryBulbTemperature':
… 'first', 'averageObscuration': 'mean'})
343
344 # further average both obscuration and daily rainfall by calendar day
345 by_date = df.groupby([df.index.month,
… df.index.day]).agg({'DailyMaximumDryBulbTemperature': 'mean',
… 'averageObscuration': 'mean'})
346 by_date.index = by_date.index.rename(['month', 'day'])
347 by_date.columns = ['mean daily max temp (F)', 'mean obscuration']
348 by_date = by_date.reset_index()
349
350 # filter out clearish days
351 by_date = by_date[by_date['mean obscuration'] <= 3.5] # 3.5 is a
… conservative cut-off for a clear day: there are at worst "scattered
… clouds"
352
353 # sort by ascending rainfall
354 by_date = by_date.sort_values(by='mean daily max temp (F)')
355
356 # add a date column to serve as the index
357 by_date['date'] = by_date.apply(lambda x: date(year=1,
… month=int(x['month']), day=int(x['day'])), axis=1)
358 by_date
359
360
361 # plot
362 b = sns.barplot(x='date', y='mean daily max temp (F)', color='skyblue',
… data=by_date)
363 b.set_title('The 36 Clearish Days Ordered by Decade Average Daily Max
… Tempearture')
364 plt.xticks(rotation=90)
365 plt.show()
366
367 sns.set()
368 plt.figure(figsize=(12, 6))
369
370 x = df
371 bool_index = (x.index.year >= 2014) & (x.index.year <= 2018) # only these
… years have data for all days of the year
372 x = x[bool_index]
373
```

```
374 mean_obsc = x.groupby([x.index.year,
… x.index.dayofyear])['averageObscuration'].mean().rolling(14).mean()
375
376 mean_obsc_decade =
… x.groupby([x.index.dayofyear])['averageObscuration'].mean().rolling(14).
… mean().plot(label='decade average', linestyle='--')
377
378 mean_obsc = mean_obsc.unstack(level=0)
379
380 # plot
381 for col in mean_obsc.columns:
382     plt.plot(mean_obsc[col], label=str(col), alpha=.4)
383 plt.legend()
384 plt.title('Moving Two-Week Average of Sky Obscuration Exhibits Annual
… Seasonality')
385 plt.xlabel('Day of Year')
386 plt.ylabel('Obscuration')
387 plt.show()
388
389 fig, ax1 = plt.subplots(figsize=(12, 6))
390 sns.set_style('white')
391
392 x = df
393 bool_index = (x.index.year >= 2014) & (x.index.year <= 2018) # only these
… years have data for all days of the year
394 x = x[bool_index]
395 x = x[['averageObscuration', 'DailyMaximumDryBulbTemperature']]
396
397 days = x.groupby([x.index.year,
… x.index.dayofyear]).mean().rolling(45).mean()
398 days.index = days.index.rename(['year', 'day'])
399 days = days.reset_index()
400 days = days.drop(['year', 'day'], axis=1)
401
402 # plot
403 [ax1.axvline(x, color='g', linestyle='--') for x in [y*365 for y in
… range(5)]] # show year starts
404
405 ax1.plot(days['averageObscuration'], label='mean decade obscuration')
406 ax1.set_xlabel('Day in 2014-2018 Time Period')
407 ax1.set_ylabel('Obscuration (45-Day Moving Average)')
408 ax1.legend(loc='upper left')
```

```
409 ax1.axhline(4, color='r')
410 ax2 = ax1.twinx() # share x axis, use two separate y axes on left and
    … right sides
411 ax2.plot(days['DailyMaximumDryBulbTemperature'], label='mean decade daily
    … max temperature (F)', alpha=0.2)
412 ax2.legend(loc='lower right')
413 ax2.set_ylabel('Temperature (45-Day Moving Average) (F)')
414 plt.title('Annual Temperature and Obscuration Seasonalities Roughly
    … Align')
415 plt.show()
416
417 fig, ax1 = plt.subplots(figsize=(12, 6))
418
419 x = df
420 bool_index = (x.index.year >= 2014) & (x.index.year <= 2018) # only these
    … years have data for all days of the year
421 x = x[bool_index]
422 x =
    … x.groupby(x.index.dayofyear).mean()['HourlyRelativeHumidity'].rolling(14).
    … mean()
423
424 plt.plot(x, label='hourly humidity (2-week rolling average)')
425 [plt.axvline(x, linestyle='--', color='g') for x in [y*92 for y in
    … range(1, 4)]]
426 plt.xlabel('Calendar Day of the Year')
427 plt.ylabel('decade average % humidity (2-week moving average)')
428 plt.title('2014-2018 Averaged Annual Humidity Fluctuates between about 72%
    … and 84%')
429 plt.show()
430
431 fig, ax1 = plt.subplots(figsize=(12, 6))
432 sns.set_style('whitegrid')
433
434 x = df
435 bool_index = (x.index.year >= 2014) & (x.index.year <= 2018) # only these
    … years have data for all days of the year
436 x = x[bool_index]
437 x = x.groupby([x.index.hour]).mean()['HourlyRelativeHumidity']
438 x.head()
439
440 sns.lineplot(data=x)
441 plt.xticks(range(24))
```

```
442 plt.xlabel('Hour of the Day')
443 plt.ylabel('% Humidity (Decade Average)')
444 ax1.xaxis.grid(which="major")
445 plt.axvline(10, color='r', linestyle='--')
446 plt.axvline(16, color='r', linestyle='--')
447 plt.title('Lower Obscuration (10 AM - 4 PM) Correlates with Humidity Below
… 70%')
448 plt.show()
449
450 # obscuration risk ratio
451 df_risk = pd.DataFrame()
452 x = df
453 # group boolean obscuration by date
454 x = df[(df.index.hour >= 10) & (df.index.hour <= 16)] # get 10 AM to 4 PM
455 x = x.groupby(x.index.date).mean() # obscuration averaged by date
456 x['is_obscured'] = x.averageObscuration >= 4.0 # add yes or no obscuration
… column
457 # recover datetime index
458 x.index = pd.Series(x.index, dtype='datetime64[ns]')
459
460 # drop Feb 29
461 mask = (x.index.day == 29) & (x.index.month == 2)
462 x = x.loc[~mask]
463 x.shape
464
465 # group yes counts by calendar day
466 x = x.groupby([x.index.month, x.index.day]).sum()
467 x.shape
468
469 def calculate_relative_obscuration_risk(row):
470     i = row.name
471     month = i[0]
472     day = i[1]
473     x_month = x.loc[month]
474     x_not = x_month[x_month.index != day] # exclude the day in question
… from the rest of the month
475     month_mean_obscured = x_not.is_obscured.mean() # average obscured days
… for rest of month
476     ratio = row.is_obscured/month_mean_obscured # compare with day in
… question's obscured days
477     return ratio
478
```

```
479 x['ratio'] = x.apply(calculate_relative_obscuration_risk, axis=1)
480 for i, month in enumerate(['Jan', 'Feb', 'March', 'April', 'May', 'June',
… 'July', 'August', 'Sept', 'Oct', 'Nov', 'Dec']):
481     plt.figure(figsize=(12, 5))
482     month_index = i + 1
483     month_frame = x[x.index.get_level_values(0) == month_index]
484     sns.barplot(x=month_frame.index, y=month_frame.ratio,
… data=month_frame, color='skyblue')
485     percentiles = np.array([0, 25, 50, 75, 100])
486     percentiles_ratio = np.percentile(month_frame['ratio'], percentiles)
487     [plt.axhline(x, linestyle='--', color='r') for x in percentiles_ratio]
488     plt.xlabel('Date in ' + month)
489     plt.xticks(rotation=60)
490     plt.ylabel('Relative Obscuration Risk Ratio')
491     plt.show()
492
493 # write out obscuration risk to csv
494 x = x.reset_index()
495 x['ratio'].to_csv('calendar_obscuration_risk.csv')
496
```

```
# time series forecasting for Monterey Airport Weather Almanacs
# Jeff Trevino, 2019
from datetime import datetime
from random import seed, random

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller, coint,
…arma_order_select_ic
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.arima_model import ARMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.holtwinters import ExponentialSmoothing

from fbprophet import Prophet

# pandas settings
pd.set_option('display.max_columns', 125) # csv contains 124 columns
pd.set_option('display.max_rows', 4000) # display more rows
pd.plotting.register_matplotlib_converters()

df = pd.read_csv('cleaned_df.csv', parse_dates=['datetime'],
…index_col=['datetime'])

x = df
bool_index = (x.index.hour >= 10) & (x.index.hour <= 16) # consider the
…clearest and driest part of each day
x = x[bool_index]

obsc = x['averageObscuration'].resample(rule='D').mean().dropna()
obsc = obsc - obsc.min() # avoid negative predictions by subtracting the
…minimum value
hum = x['HourlyRelativeHumidity'].resample(rule='D').mean().dropna()
temp =
…x['DailyMaximumDryBulbTemperature'].resample(rule='D').last().dropna()

# we need the 2019 data to assess predictions later
obsc_all = obsc[obsc.index.year == 2019]
```

```
38 hum_all = hum[hum.index.year == 2019]
39 temp_all = temp[temp.index.year == 2019]
40
41 sns.set()
42
43 bool_index = (obsc.index.year >= 2014) & (obsc.index.year <= 2018)
44 obsc = obsc[bool_index]
45 bool_index = (temp.index.year >= 2010) & (temp.index.year <= 2018)
46 temp = temp[bool_index]
47 bool_index = (hum.index.year >= 2010) & (hum.index.year <= 2018)
48 hum = hum[bool_index]
49
50 plt.figure(figsize=(12, 6))
51 plt.plot(obsc, color='b', alpha=0.2)
52 obsc.rolling(14).mean().plot()
53 obsc.rolling(14).var().plot(alpha=0.5)
54 plt.xlabel('date')
55 plt.ylabel('obscuration')
56 plt.legend(('daily obscuration', 'rolling 2-week mean', 'rolling 2-week
… variance'))
57 plt.show()
58
59 plt.figure(figsize=(12, 6))
60 plt.plot(temp, color='b', alpha=0.2)
61 temp.rolling(14).mean().plot()
62 temp.rolling(14).var().plot(alpha=0.5)
63 plt.xlabel('date')
64 plt.ylabel('degrees (F)')
65 plt.legend(('daily max temp (F)', 'rolling 2-week mean', 'rolling 2-week
… variance'))
66
67 plt.figure(figsize=(12, 6))
68 plt.plot(hum, color='b', alpha=0.2)
69 hum.rolling(14).mean().plot()
70 hum.rolling(14).var().plot(alpha=0.5)
71 plt.xlabel('date')
72 plt.ylabel('humidity (%)')
73 plt.legend(('daily max temp (F)', 'rolling 2-week mean', 'rolling 2-week
… variance'))
74 plt.show()
75
76 # Dickey-Fuller tests for temperature, humidity, obscuration
```

```
77 results = adfuller(obsc)
78 print("p-value is:", results[1])
79 results = adfuller(temp)
80 print("p-value is:", results[1])
81 # conduct test
82 results = adfuller(hum)
83 print("p-value is:", results[1])
84
85 # sanity check: does a random walk have a time dependent structure?
86
87 # Generate random residuals
88 np.random.seed(0)
89 errors = np.random.normal(0, 1, 1000)
90
91 # Create AR(1) (random walk) samples for models with and without unit
… roots
92 x_unit_root = [0]
93 x_no_unit_root = [0]
94 for i in range(len(errors)):
95     x_unit_root.append(x_unit_root[-1] + errors[i])
96     x_no_unit_root.append(0.9*x_no_unit_root[-1] + errors[i]) # (0.9 isn't
… 1, so no unit root)
97
98 # Calculate Augmented Dickey-Fuller p-values
99 adfuller(x_unit_root)[1], adfuller(x_no_unit_root)[1] # good: a random
… walk is non-stationary
100
101 # autocorrelation and partial autocorrelation plots
102 plot_acf(obsc, lags=20)
103 plt.show()
104 plot_pacf(obsc, lags=20)
105 plt.show()
106
107 plot_acf(temp, lags=75)
108 plt.show()
109 plot_pacf(temp, lags=100)
110 plt.show()
111
112 plot_acf(hum, lags=25)
113 plt.show()
114 plot_pacf(hum, lags=25)
115 plt.show()
```

```
116
117 for series in [temp, obsc, hum]:
118     result = arma_order_select_ic(series)['bic_min_order']
119     print(str(series.name), ": ", result)
120
121
122 # ARMA Models
123 # Fit an ARMA model to the first simulated data
124 model = ARMA(temp, order=(3,1)) # fit to ARMA model
125 fitted = model.fit()
126
127 # Print out summary information on the fit
128 print(fitted.summary())
129
130 # Print out the estimate for the constant and for phi
131 print("The estimate of phi (and the constant) are:")
132 print(fitted.params)
133
134 # forecast the past...
135 cast = fitted.predict(start='01-01-2010', end='12-31-2018')
136 fig, ax = plt.subplots(figsize=(12, 6))
137 plt.plot(temp, label='truth')
138 plt.plot(cast, label='ARMA (3,1)')
139 plt.xlabel('date')
140 plt.ylabel('temperature (F)')
141 plt.title('ARMA Tempearture Prediction')
142 plt.legend()
143 plt.show()
144
145 forecast = fitted.forecast(31)[0]
146 # forecast Jan 2019 and measure error against observed
147 # get January of 2019 values
148 x = df
149 jan_nineteen = x[x.index.year == 2019]
150 jan_nineteen = jan_nineteen[(jan_nineteen.index.hour >= 10) &
… (jan_nineteen.index.hour <= 16)]
151 jan_nineteen =
… jan_nineteen['DailyMaximumDryBulbTemperature'].resample(rule='D').last().
… dropna()
152 jan_nineteen = jan_nineteen[jan_nineteen.index.month == 1]
153
154 # calculate error
```

```
155 rmse = np.sqrt(np.mean(np.square(forecast - jan_nineteen.values)))
156
157 # give predicted values a datetime index
158 index = pd.date_range(start='01-01-2019', end='01-31-2019')
159 jan_predicted = pd.DataFrame(forecast)
160 jan_predicted = jan_predicted.set_index(index)
161
162 # overlay predicted values with measured values
163 plt.plot(jan_nineteen, alpha=.4, label='truth, Jan 19')
164 plt.plot(jan_predicted, label='ARMA (3,1) RMSE: {:0.2f}'.format(rmse))
165 plt.xticks(rotation=60)
166 plt.legend()
167 plt.xlabel('date')
168 plt.ylabel('temperature (F)')
169 plt.title('ARMA temperature estimates for January of 2019 average 4.65
  … degrees of error')
170 plt.show()
171
172 # the same for obscuration
173 # Fit an ARMA model to the first simulated data
174 model = ARMA(obsc, order=(1,0)) # fit to ARMA model
175 fitted = model.fit()
176
177 # Print out summary information on the fit
178 print(fitted.summary())
179
180 # Print out the estimate for the constant and for phi
181 print("The estimate of phi (and the constant) are:")
182 print(fitted.params)
183
184 # forecast the past...
185 cast = fitted.predict(start='01-01-2014', end='12-31-2018')
186 fig, ax = plt.subplots(figsize=(12, 6))
187 plt.plot(obsc, label='truth')
188 plt.plot(cast, label= 'ARMA (1,0)')
189 plt.xlabel('date')
190 plt.ylabel('obscuration')
191 plt.title('ARMA Obscuration Prediction')
192 plt.legend()
193 plt.show()
194
195 forecast = fitted.forecast(31)[0]
```

```
196
197 # get January of 2019 values
198 x = df
199 jan_nineteen = x[x.index.year == 2019]
200 jan_nineteen = jan_nineteen[(jan_nineteen.index.hour >= 10) &
… (jan_nineteen.index.hour <= 16)]
201 jan_nineteen =
… jan_nineteen['averageObscuration'].resample(rule='D').last().dropna()
202 jan_nineteen = jan_nineteen[jan_nineteen.index.month == 1]
203
204 # calculate error
205 rmse = np.sqrt(np.mean(np.square(forecast - jan_nineteen.values)))
206
207 # give predicted values a datetime index
208 index = pd.date_range(start='01-01-2019', end='01-31-2019')
209 jan_predicted = pd.DataFrame(forecast)
210 jan_predicted = jan_predicted.set_index(index)
211
212 # overlay predicted values with measured values
213 plt.plot(jan_nineteen, alpha=.4, label='truth, Jan 19')
214 plt.plot(jan_predicted, label='ARMA (1,0) RMSE: {:0.2f}'.format(rmse))
215 plt.xticks(rotation=60)
216 plt.legend()
217 plt.xlabel('date')
218 plt.ylabel('obscuration')
219 plt.title('ARMA obscuration estimates for January of 2019 average error of
… 3 (nearly 50%)')
220 plt.show()
221
222 # humidity
223 # Fit an ARMA model to the first simulated data
224 model = ARMA(hum, order=(2, 2)) # fit to ARMA model
225 fitted = model.fit()
226
227 # Print out summary information on the fit
228 print(fitted.summary())
229
230 # Print out the estimate for the constant and for phi
231 print("The estimate of phi (and the constant) are:")
232 print(fitted.params)
233
234 # forecast the past...
```

```
cast = fitted.predict(start='01-01-2010', end='12-31-2018')
fig, ax = plt.subplots(figsize=(12, 6))
plt.plot(hum, label='truth')
plt.plot(cast, label='ARMA (2,2)')
plt.xlabel('date')
plt.ylabel('humidity (%)')
plt.title('ARMA Humidity Prediction')
plt.legend()
plt.show()
plt.show()

forecast = fitted.forecast(31)[0]

# get January of 2019 values
x = df
jan_nineteen = x[x.index.year == 2019]
jan_nineteen = jan_nineteen[(jan_nineteen.index.hour >= 10) &
(jan_nineteen.index.hour <= 16)]
jan_nineteen =
jan_nineteen['HourlyRelativeHumidity'].resample(rule='D').mean().dropna()
jan_nineteen = jan_nineteen[jan_nineteen.index.month == 1]

# calculate error
rmse = np.sqrt(np.mean(np.square(forecast - jan_nineteen.values)))

# give predicted values a datetime index
index = pd.date_range(start='01-01-2019', end='01-31-2019')
jan_predicted = pd.DataFrame(forecast)
jan_predicted = jan_predicted.set_index(index)

# overlay predicted values with measured values
plt.plot(jan_nineteen, alpha=.4, label='truth, Jan 19')
plt.plot(jan_predicted, label='ARMA (2, 2) RMSE: {:0.2f}'.format(rmse))
plt.xticks(rotation=60)
plt.legend()
plt.xlabel('date')
plt.ylabel('humidity (%)')
plt.title('ARMA humidity estimates for Jan 2019 average 15% error')
plt.show()

# naive season-trend decomposition
results = seasonal_decompose(obsc, model='additive', freq=365)
```

```
275 results.plot()
276 plt.show()
277
278 df_results = adfuller(results.trend.dropna())
279 print("trend df p-value is:", df_results[1]) # trend isn't stationary (it
… trends)
280
281 df_results = adfuller(results.seasonal.dropna())
282 print("seasonal df p-value is:", df_results[1]) # seasonality is
283
284 df_results = adfuller(results.resid.dropna())
285 print("residuals p-value is:", df_results[1]) # so are residuals
286
287 results = seasonal_decompose(temp, model='additive', freq=365)
288 results.plot()
289 plt.show()
290
291 df_results = adfuller(results.trend.dropna())
292 print("trend df p-value is:", df_results[1]) # trend isn't stationary (it
… trends)
293
294 df_results = adfuller(results.seasonal.dropna())
295 print("seasonal df p-value is:", df_results[1]) # seasonality is
296
297 df_results = adfuller(results.resid.dropna())
298 print("residuals df p-value is:", df_results[1]) # so are residuals
299
300 # Holt-Winters Seasonal Smoothing
301 # separate data into train and test sets
302 train = temp[:-365]
303 test = temp.iloc[-365:]
304 # initialize models
305 model1 = ExponentialSmoothing(train, trend='add', seasonal='add',
… seasonal_periods=365)
306 model2 = ExponentialSmoothing(train, trend='add', seasonal='add',
… seasonal_periods=365, damped=True)
307 model3 = ExponentialSmoothing(train, trend='add', seasonal='mul',
… seasonal_periods=365, damped=True)
308 # fit models to data
309 fit1 = model1.fit()
310 cast1 = fit1.forecast(365)
311 fit2 = model2.fit()
```

```
cast2 = fit2.forecast(365)
fit3 = model3.fit()
cast3 = fit3.forecast(365)
# calculate error
sse1 = np.sqrt(np.mean(np.square(test.values - cast1.values)))
sse2 = np.sqrt(np.mean(np.square(test.values - cast2.values)))
sse3 = np.sqrt(np.mean(np.square(test.values - cast3.values)))
# plot
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(train.index[-365:], train.values[-365:])
ax.plot(test.index, test.values, label='truth',color='b', alpha=.5);
ax.plot(test.index, cast1, color='r', label="add undamped (RMSE={:0.2f},
…AIC={:0.2f})".format(sse1, fit1.aic));
ax.legend();
ax.set_xlabel('date')
ax.set_ylabel('degrees (F)')
ax.set_title("Holt-Winter's Seasonal Smoothing Temperature Forecast");
plt.show()

fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(train.index[-365:], train.values[-365:])
ax.plot(test.index, test.values, label='truth',color='b', alpha=.5);
ax.plot(test.index, cast2, color='g', label="add damped (RMSE={:0.2f},
…AIC={:0.2f})".format(sse2, fit2.aic));
ax.legend();
ax.set_xlabel('date')
ax.set_ylabel('degrees (F)')
ax.set_title("Holt-Winter's Seasonal Smoothing Temperature Forecast");
plt.show()

fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(train.index[-365:], train.values[-365:])
ax.plot(test.index, test.values, label='truth',color='b', alpha=.5);
ax.plot(test.index, cast3, color='black', label="mult damped
…(RMSE={:0.2f}, AIC={:0.2f})".format(sse3, fit3.aic));
ax.legend();
ax.set_xlabel('date')
ax.set_ylabel('degrees (F)')
ax.set_title("Holt-Winter's Seasonal Smoothing Temperature Forecast");
plt.show()

# separate data into train and test sets
```

```
351 train = obsc.iloc[:-365]
352 test = obsc.iloc[-365:]
353 # initialize models
354 model1 = ExponentialSmoothing(train, trend='add', seasonal='add',
  … seasonal_periods=365)
355 model2 = ExponentialSmoothing(train, trend='add', seasonal='add',
  … seasonal_periods=365, damped=True)
356
357 # fit models to data
358 fit1 = model1.fit()
359 cast1 = fit1.forecast(365)
360 cast1 = cast1 - cast1.min()
361
362 # failing as all NaNs for unknown reason
363 # fit2 = model2.fit()
364 # cast2 = fit2.forecast(365)
365 # cast2 = cast2 - cast2.min()
366 # cast2
367
368 # calculate error
369 sse1 = np.sqrt(np.mean(np.square(test.values - cast1.values)))
370 # sse2 = np.sqrt(np.mean(np.square(test.values - cast2.values))) # fails
  … as NaN
371
372 # plot
373 fig, ax = plt.subplots(figsize=(12, 6))
374 ax.plot(train.index[-365:], train.values[-365:])
375 ax.plot(test.index, test.values, label='truth',color='b', alpha=.5);
376 ax.plot(test.index, cast1, color='r', label="add undamped (RMSE={:0.2f},
  … AIC={:0.2f})".format(sse1, fit1.aic));
377 ax.legend();
378 ax.set_xlabel('date')
379 ax.set_ylabel('obscuration')
380 ax.set_title("Holt-Winter's Seasonal Smoothing Obscuration Forecast");
381 plt.show()
382
383 # failing
384 # fig, ax = plt.subplots(figsize=(12, 6))
385 # ax.plot(train.index[-365:], train.values[-365:])
386 # ax.plot(test.index, test.values, label='truth',color='b', alpha=.5);
387 # ax.plot(test.index, cast2, color='g', label="add damped (RMSE={:0.2f},
  … AIC={:0.2f})".format(sse2, fit2.aic));
```

```
# ax.legend();
# ax.set_xlabel('date')
# ax.set_ylabel('obscuration')
# ax.set_title("Holt-Winter's Seasonal Smoothing Obscuration Forecast");
# plt.show()

# separate data into train and test sets
train = hum[:-365]
test = hum.iloc[-365:]

# initialize models
model1 = ExponentialSmoothing(train, trend='add', seasonal='add',
…seasonal_periods=365)
model2 = ExponentialSmoothing(train, trend='add', seasonal='add',
…seasonal_periods=365, damped=True)
model3 = ExponentialSmoothing(train, trend='add', seasonal='mul',
…seasonal_periods=365, damped=True)

# fit models to data
fit1 = model1.fit()
cast1 = fit1.forecast(365)
fit2 = model2.fit()
cast2 = fit2.forecast(365)
fit3 = model3.fit()
# cast3 = fit3.forecast(365) failing as NaNs
# cast3

# calculate error
sse1 = np.sqrt(np.mean(np.square(test.values - cast1.values)))
sse2 = np.sqrt(np.mean(np.square(test.values - cast2.values)))
sse3 = np.sqrt(np.mean(np.square(test.values - cast3.values)))

# plot
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(train.index[-365:], train.values[-365:])
ax.plot(test.index, test.values, label='truth',color='b', alpha=.5);
ax.plot(test.index, cast1, color='r', label="add undamped (RMSE={:0.2f},
…AIC={:0.2f})".format(sse1, fit1.aic));
ax.legend();
ax.set_xlabel('date')
ax.set_ylabel('degrees (F)')
ax.set_title("Holt-Winter's Seasonal Smoothing Humidity Forecast");
```

```
426 plt.show()
427
428 fig, ax = plt.subplots(figsize=(12, 6))
429 ax.plot(train.index[-365:], train.values[-365:])
430 ax.plot(test.index, test.values, label='truth',color='b', alpha=.5);
431 ax.plot(test.index, cast2, color='g', label="add damped (RMSE={:0.2f},
  … AIC={:0.2f})".format(sse2, fit2.aic));
432 ax.legend();
433 ax.set_xlabel('date')
434 ax.set_ylabel('degrees (F)')
435 ax.set_title("Holt-Winter's Seasonal Smoothing Humidity Forecast");
436 plt.show()
437
438 # failing as NaNs
439 # fig, ax = plt.subplots(figsize=(12, 6))
440 # ax.plot(train.index[-365:], train.values[-365:])
441 # ax.plot(test.index, test.values, label='truth',color='b', alpha=.5);
442 # ax.plot(test.index, cast3, color='black', label="mult damped
  … (RMSE={:0.2f}, AIC={:0.2f})".format(sse3, fit3.aic));
443 # ax.legend();
444 # ax.set_xlabel('date')
445 # ax.set_ylabel('degrees (F)')
446 # ax.set_title("Holt-Winter's Seasonal Smoothing Humidity Forecast");
447 # plt.show()
448
449 # calculate index for use in all predictions
450 forecast_index = pd.date_range(start='01-01-2019', end='12-31-2019')
451
452 # set training data
453 train = temp
454
455 # initialize models
456 temp_model = ExponentialSmoothing(train, trend='add', seasonal='mul',
  … seasonal_periods=365, damped=True)
457
458 # fit models to data
459 fit_temp = model3.fit()
460 # temp_cast = fit_temp.forecast(365) forecasting NaNs
461
462 # broken
463 # # plot
464 # fig, ax = plt.subplots(figsize=(12, 6))
```

```
# ax.plot(forecast_index, temp_cast, color='r', label="mult
…damped".format(sse1, fit1.aic));
# ax.legend();
# ax.set_xlabel('date')
# ax.set_ylabel('degrees (F)')
# ax.set_title("Holt-Winter's Seasonal Smoothing Temperature Forecast");
# plt.show()

# set training data
train = obsc
# initialize model
model = ExponentialSmoothing(train, trend='add', seasonal='add',
…seasonal_periods=365, damped=True)
# fit models to data
obsc_fit = model.fit()
obsc_cast = obsc_fit.forecast(365)
obsc_cast = obsc_cast - obsc_cast.min()
# plot
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(forecast_index, obsc_cast, color='r', label="add
…damped".format(sse1, fit1.aic));
ax.legend();
ax.set_xlabel('date')
ax.set_ylabel('obscuration')
ax.set_title("Holt-Winter's Seasonal Smoothing Obscuration Forecast");
plt.show()

# set the training data
train = hum

# initialize models
model = ExponentialSmoothing(train, trend='add', seasonal='add',
…seasonal_periods=365, damped=True)

# fit models to data
fit = model.fit()
hum_cast = fit.forecast(365)

# plot
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(forecast_index, hum_cast, color='r', label="add damped");
ax.legend();
```

```
503 ax.set_xlabel('date')
504 ax.set_ylabel('humidity (%)')
505 ax.set_title("Holt-Winter's Seasonal Smoothing Humidity Forecast");
506 plt.show()
507
508 # Facebook Prophet
509 x = df
510 bool_index = (x.index.hour >= 10) & (x.index.hour <= 16) # consider the
    … clearest and driest part of each day
511 x = x[bool_index]
512
513 obsc = x['averageObscuration'].resample(rule='D').mean().dropna()
514 obsc = obsc - obsc.min() # avoid negative predictions by subtracting the
    … minimum value
515 hum = x['HourlyRelativeHumidity'].resample(rule='D').mean().dropna()
516 temp =
    … x['DailyMaximumDryBulbTemperature'].resample(rule='D').last().dropna()
517
518 # we need the 2019 data to assess predictions later
519 obsc_all = obsc[obsc.index.year == 2019]
520 hum_all = hum[hum.index.year == 2019]
521 temp_all = temp[temp.index.year == 2019]
522
523 # seems like obscuration measurement changed around 2014; discard 2010
    … through 2014
524 obsc = obsc[obsc.index.year >= 2014]
525
526 # make dataframes fbprophet likes
527 def make_prophet_dataframe_from_series(series):
528     frame = pd.DataFrame(series).reset_index()
529     frame.columns = ['ds', 'y']
530     return frame
531
532     # make prophet frames
533 fb_temp = make_prophet_dataframe_from_series(temp)
534 fb_temp.tail()
535
536 # fit the model
537 m = Prophet()
538 m.fit(fb_temp)
539
540 # make future column
```

```
future = m.make_future_dataframe(periods=365)
future.tail()

# predict
temp_forecast = m.predict(future)
temp_forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()

# plot predictions
fig1 = m.plot(temp_forecast)

# make prophet frames
fb_obsc = make_prophet_dataframe_from_series(obsc)
fb_obsc.tail()

# fit the model
m = Prophet()
m.fit(fb_obsc)

# make future column
future = m.make_future_dataframe(periods=365)

# predict
obsc_forecast = m.predict(future)

# plot predictions
fig1 = m.plot(obsc_forecast)

# make prophet frames
fb_hum = make_prophet_dataframe_from_series(hum)
fb_hum.tail()

# fit the model
m = Prophet()
m.fit(fb_hum)

# make future column
future = m.make_future_dataframe(periods=365)

# predict
hum_forecast = m.predict(future)

# plot predictions
```

```
583 fig1 = m.plot(hum_forecast)
584
585 # plot model components
586 fig2 = m.plot_components(hum_forecast)
587
588 # compare prophet predictions to first three months of 2019
589 # get 2019 values
590 temp_nineteen = temp_all[temp_all.index.year == 2019]
591
592 # get estimates
593 temp_guess_df = pd.DataFrame(temp_forecast['yhat'][:90])
594 i = pd.date_range(start='01/01/19', end='03/31/19')
595 temp_guess_df = temp_guess_df.set_index(i)
596
597 # calculate error
598 temp_rmse = np.sqrt(np.mean(np.square(temp_guess_df.values -
  … temp_nineteen.values)))
599 temp_rmse
600
601 # overlay predicted values with measured values
602 plt.plot(temp_nineteen, alpha=.4, label='truth, 2019')
603 plt.plot(temp_guess_df, label='prophet RMSE: {:0.2f}'.format(temp_rmse))
604 plt.xticks(rotation=60)
605 plt.legend()
606 plt.xlabel('date')
607 plt.ylabel('temperature (F)')
608 plt.title('Prophet temperature estimates for Jan 2019 outperform
  … Holt-Winters smoothing')
609 plt.show()
610
611 # get 2019 values
612 obsc_nineteen = obsc_all[obsc_all.index.year == 2019]
613 obsc_nineteen += np.min(obsc_nineteen)
614 len(obsc_nineteen)
615
616 # get estimates
617 guess_df = pd.DataFrame(obsc_forecast['yhat'][:90])
618 i = pd.date_range(start='01/01/19', end='03/31/19')
619 guess_df = guess_df.set_index(i)
620
621 # calculate error
622 obsc_rmse = np.sqrt(np.mean(np.square(guess_df.values -
```

```
622… obsc_nineteen.values)))
623 obsc_rmse
624
625 # overlay predicted values with measured values
626 plt.plot(obsc_nineteen, alpha=.4, label='truth, 2019')
627 plt.plot(guess_df, label='prophet RMSE: {:0.2f}'.format(obsc_rmse))
628 plt.xticks(rotation=60)
629 plt.legend()
630 plt.xlabel('date')
631 plt.ylabel('obscuration')
632 plt.title('Prophet obscuration estimates for Jan 2019 outperforms
  … Holt-Winteres smoothing')
633 plt.show()
634
635 # get 2019 values
636 hum_nineteen = hum_all[temp_all.index.year == 2019]
637
638 # get estimates
639 guess_df = pd.DataFrame(hum_forecast['yhat'][:90])
640 i = pd.date_range(start='01/01/19', end='03/31/19')
641 guess_df = guess_df.set_index(i)
642
643 # calculate error
644 rmse = np.sqrt(np.mean(np.square(guess_df.values - hum_nineteen.values)))
645 rmse
646
647 # overlay predicted values with measured values
648 plt.plot(hum_nineteen, alpha=.4, label='truth, 2019')
649 plt.plot(guess_df, label='prophet RMSE: {:0.2f}'.format(rmse))
650 plt.xticks(rotation=60)
651 plt.legend()
652 plt.xlabel('date')
653 plt.ylabel('humidity (%)')
654 plt.title('Prophet humidity estimates for Jan 2019 perform slightly under
  … Holt-Winters smoothing')
655 plt.show()
656
657 x = obsc_forecast
658 x = x.set_index('ds')
659 x.index.name = 'date'
660 obsc_predictions = x['yhat']
661
```

```
x = temp_forecast
x = x.set_index('ds')
x.index.name = 'date'
temp_predictions = x['yhat']

x = hum_forecast
x = x.set_index('ds')
x.index.name = 'date'
hum_predictions = x['yhat']

nineteen_hat = pd.DataFrame({'temp': temp_predictions, 'hum':
hum_predictions, 'obsc': obsc_predictions})
nineteen_hat = nineteen_hat[nineteen_hat.index.year >= 2014]
sns.heatmap(nineteen_hat.isnull(), cbar=False)
nineteen_hat.tail()

nineteen_hat.to_csv('predictions.csv') # export fbrophet predictions

```

```python
# event posting code for Monterey Airport Weather Almanacs
# Jeff Trevino, 2019
# imports
from __future__ import print_function
import datetime
import pickle
import os.path

from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
import pandas as pd

# If modifying these scopes, delete the file token.pickle.
SCOPES = ['https://www.googleapis.com/auth/calendar']

risk_ratios = pd.read_csv('calendar_obscuration_risk.csv', index_col=0, names=['risk_ratio'])
risk_ratios.index = risk_ratios.index + 1

predictions = pd.read_csv('predictions.csv', index_col=0, names=['temp', 'hum', 'obsc'], header=0, parse_dates=True)
mask = (predictions.index.date == 29) & (predictions.index.month == 2) # remove leap years
predictions = predictions[~mask]

# create day of year column
predictions['dayofyear'] = predictions.index.dayofyearx = predictions
x['dayofyear'] = x['dayofyear'].apply(lambda day: day - 1 if day > 60 else day)

# a helper function looks up the risk ratio by the day of the year
def get_risk_ratio_by_dayofyear(row):
    return risk_ratios.loc[row['dayofyear']]
# add risk_ratio to prediction frame
predictions['obscuration_risk_ratio'] = predictions.apply(get_risk_ratio_by_dayofyear, axis=1)
# drop dayof year
predictions = predictions.drop(columns=['dayofyear'])
x = predictions

# calendar setup
```

```
38 creds = None
39 # The file token.pickle stores the user's access and refresh tokens, and
… is
40 # created automatically when the authorization flow completes for the
… first
41 # time.
42 if os.path.exists('token.pickle'):
43     with open('token.pickle', 'rb') as token:
44         creds = pickle.load(token)
45
46          # If there are no (valid) credentials available, let the user log
… in.
47 if not creds or not creds.valid:
48     if creds and creds.expired and creds.refresh_token:
49         creds.refresh(Request())
50     else:
51         flow = InstalledAppFlow.from_client_secrets_file(
52             'credentials.json', SCOPES)
53         creds = flow.run_local_server(port=0)
54     # Save the credentials for the next run
55     with open('token.pickle', 'wb') as token:
56         pickle.dump(creds, token)
57
58 service = build('calendar', 'v3', credentials=creds) # uncomment to sign
… in
59
60 # get a calendar list
61 page_token = None
62 while True:
63   calendar_list =
… service.calendarList().list(pageToken=page_token).execute()
64   for calendar_list_entry in calendar_list['items']:
65     print(calendar_list_entry['accessRole'], ":",
… calendar_list_entry['summary'])
66   page_token = calendar_list.get('nextPageToken')
67   if not page_token:
68     break
69
70     # get calendar id
71 el_cid = calendar_list['items'][4]['id']
72
73 def build_daily_string(row):
```

```
74      the_string = ''
75      the_string += 'temp: ' + '{:.0f}'.format(row['temp']) + ' F ' + '\n'
76      the_string += 'hum: ' + '{:.0f}'.format(row['hum']) + '%' + '\n'
77      the_string += 'obsc(0-8): ' + '{:.0f}'.format(row['obsc']) + '\n'
78      the_string += 'ORR: ' + '{:.2f}'.format(row['obscuration_risk_ratio'])
… + '\n'
79      return the_string
80
81  def make_event_body(date_string, the_string):
82      body = {'summary': the_string,
83              'location': 'Monterey Airport',
84              'description': 'A weather prediction for event planners',
85              'start': {
86                  'date': date_string
87              },
88              'end': {
89                  'date': date_string
90              },
91             }
92      return body
93
94  def make_event_metadata(row, eid, cid, service):
95      # define a patch (the info to add in custom fields)
96      body = {
97        'extendedProperties': {
98          'private': {
99              'temperature': '{:.0f}'.format(row['temp']) + ' F ',
100             'humidity': '{:.0f}'.format(row['hum']) + '%',
101             'obscuration(0-8)': '{:.0f}'.format(row['obsc']),
102             'obscuration risk ratio':
… '{:.2f}'.format(row['obscuration_risk_ratio'])
103         }
104       }
105     }
106
107
108 def event_from_row(row, cid):
109     date_string = str(index.date())
110     event_summary_string = build_daily_string(row)
111     mr_body = make_event_body(date_string, event_summary_string)
112     event = service.events().insert(calendarId=cid,
… body=mr_body).execute()
```

```
#     print('Event created: %s' % (event.get('htmlLink'))) # maybe add metadata in future
#     so_meta = make_event_metadata(row, event['id'], cid, service)
#     service.events().patch(calendarId=cid, eventId=event['id'], body=so_meta).execute()

mask = (predictions.index.year >= 2019) & (predictions.index.month >= 9)
predictions = predictions[mask]

# get ahold of the end of 2019
rest_of_nineteen = predictions.loc['2019-09-01':'2019-12-31']

# post predictions for rest of 2019 to calendar
for index, row in rest_of_nineteen.iterrows():
    date_string = str(index.date())
    event_summary_string = build_daily_string(row)
    mr_body = make_event_body(date_string, event_summary_string)
    service.events().insert(calendarId=el_cid, body=mr_body).execute()
```