Appendix A

Finishing Column Region

In this section, we are going to finish the management of column *Region*. First, we make a copy of the data frame we were working with:

```
wildf_dfv21 = wildf_dfv2.copy()
```

To deal with the missing names of states we are going to create a dictionary in which key-value pairs are the name of the state and its abbreviation respectively:

```
us_states={'Alabama':'AL', 'Arizona':'AZ', 'Arkansas':'AR',
           'California':'CA', 'Colorado':'CO', 'Connecticut':'CT',
           'Delaware':'DE', 'Florida':'FL', 'Georgia':'GA',
           'Hawaii':'HI', 'Idaho':'ID', 'Illinois':'IL',
           'Indiana':'IN', 'Iowa':'IA', 'Kansas':'KS',
           'Kentucky':'KY', 'Louisiana':'LA', 'Maine':'ME',
           'Maryland':'MD', 'Massachusetts':'MA', 'Michigan':'MI',
           'Minnesota':'MN', 'Mississippi':'MS', 'Missouri':'MO',
           'Montana':'MT', 'Nebraska':'NE', 'Nevada':'NV',
           'New Hampshire':'NH', 'New Jersey':'NJ',
           'New Mexico':'NM', 'New York':'NY', 'North Carolina':'NC',
           'North Dakota':'ND', 'Ohio':'OH', 'Oklahoma':'OK',
           'Oregon':'OR', 'Pennsylvania':'PA', 'Rhode Island':'RI',
           'South Carolina':'SC', 'South Dakota':'SD',
           'Tennessee':'TN', 'Texas':'TX', 'Utah':'UT',
           'Vermont':'VT', 'Virginia':'VA', 'Washington':'WA',
           'West Virginia':'WV', 'Wisconsin':'WI', 'Wyoming':'WY'}
```

To retrieve the information conveyed in the column *Location* we are going to write a custom function to extract the name of the state. Figure A.1 shows the custom function *loc_to_state* which takes as arguments the data frame and the column *Location*. Lines 2 to 5 define the variables which are going to be explained as we go over the function. Line 6 initializes the *for* loop to iterate over column *Location*. Line 7 tests if the element at position *i* in column *Region* is a string. If it is a string it means that the cell is not empty, i.e., not an **NaN** value, and in that case the **if not** returns **False** and the variable *i* is increment by 1 in line 34 to keep both columns synchronized. If line 7 returns **True** it means that the cell is vacant so we can populate it.

Line 8 tests this time if the element of column *Location*, *sentence*, is a string, if **False** then in line 32 the variable *i* is incremented by 1. If line 8 returns **True** then in line 9 the **findall()** method returns a list of the occurrences of the regular expression pattern '\w+', in this case, the whole

```
def loc_to_state(df, col):
    i=0
    flag=0
    list_count = 0
    wordup = ''
    for sentence in df[col]:
        if not isinstance(df.at[i,'Region'], str):
            if isinstance(sentence, str):
                word_list = re.findall(r'\w+', sentence)
                word_list_len = len(word_list)
                for word in word_list:
                    list_count += 1
                    if len(word)>2:
                        wordup = word.upper()
                    else:
                        wordup = word
                    for st_name, st_abbrev in us_states.items():
                        upper_st = st_name.upper()
                        if (wordup == upper_st or wordup == st_abbrev):
                            df.at[i,'Region'] = st_name
                            i+=1
                            flag = 1
                            break
                    if (list_count == word_list_len and flag ==0):
                        list_count = 0
                        i+=1
                    elif(flag == 1):
                        list_count = 0
                        flag = 0
                        break
            else:
                i+=1
        else:
            i+=1
```

Figure A.1: Function loc_to_state.

string, and copies the value into variable *word_list*. From now on we proceed with the explanation of the function with an example.

Let's suppose the string 'Lake Hughes, Unincorporated LA County' is copied into *word_list*, because this variable is a list this is what we get if we print it:

```
print(word_list)
```

```
['Lake', 'Hughes', 'Unincorporated', 'LA', 'County']
```

In line 10 the length of *word_list*, 5, is copied into variable *word_list_len*. In line 11 we iterate through *word_list*. Line 12 increments the control variable *list_count* by 1, soon we are going to know why. In line 13 if the length of the string contained in *word* is greater than 2 then in line 14 it will be converted to all capitals with method **upper()**, in the case of the first item in *word_list* 'Lake' translates to 'LAKE'. If the length of the item is not greater than 2 then the item will be copied as-is into variable *wordup* in line 16.

Now is time to check if the word in the list is a name of a state or an abbreviation. Line 17 iterates through each key-value pair in the dictionary. Line 18 sets the name of the state to all capitals for comparison with the item of the list. Line 19 tests for a match, if **True** then in line 20 the name of the state is copied into the corresponding cell of the column *Region* and the control variable *i* is incremented by 1 in line 21. Line 22 sets the variable *flag* to 1 and in line 23 we drop out of the *for* loop, this way we don't keep on iterating through the dictionary wasting resources.

The **if** condition in line 24 tests if we reached the end of the list, the control variable *list_count* tracks the list while it is traversed, and if there was not any match (flag = 0), if **True** then in line 25 *list_count* is reset and in line 26 *i* is incremented by one to go to the next row. If line 24 returns **False** then we check if the flag is set to 1 in line 27, if **True** then in line 28 *list_count* is reset as well as the flag in line 29, and in line 30 we drop out of the *for* loop that operates on the list. After this the outer *for* loop goes to the next element of the column *Location*.

To illustrate this suppose the *for* loop in line 11 retrieves the item 'LA' , because this string is all capitals is copied as-is in variable *wordup* in line 16. When tested for a match in line 19 it returns **True** because 'LA' corresponds to the key 'Los Angeles' in the dictionary. Then in line 20 the name of the state is copied into the corresponding cell in the data frame and the flag is set to 1 and we drop out of the *for* loop. Line 24 returns **False** so we jump to line 27. Since the flag is set to 1, then *list_count* and the flag are reset. With the command **break** we stop iterating through the list so that 'County' won't be tested. After this, the outer *for* loop steps in the next row of column *Location*.

Upon calling the function

```
loc_to_state(wildf_dfv21, 'Location')
```

the missing names of states should appear in column *Region* whenever there was a match in column *Location*. A caveat, one of the four of the subcardinal points, Northeast (NE), concurs with the abbreviation of the state of Nebraska as shown below:

```
wildf_dfv21.loc[18,['Location','Region']]
```

```
Location     NE of New River
Region                Nebraska
Name:  18, dtype:  object
```

Since there are several rivers named New River, a quick look on the internet using the information conveyed in column *Incident*, and because is our lucky day, showed there was a fire northeast of New River, Arizona. We bet for this entry and proceed to change the name of the state on the only entry with the abbreviation NE:

```
wildf_dfv21.at[18, 'Region'] ='Arizona'
```

Now, the code shown in figure A.2 shows all rows related to column *Location* where the values in column *Region* are missing.

```
1 i = 0
2 indexes = list()
3 for item, name in zip(wildf_dfv21['Region'], wildf_dfv21['Location']):
4     if (isinstance(item, float) and not isinstance(name, float)):
5         indexes.append(wildf_dfv21.index[wildf_dfv21['Location'] == name].tolist())
6         print(indexes[i][0], '->', name)
7         i += 1
```

Figure A.2: Code for listing the rows of column Location *that correspond to missing values of column* Region.

The output of the last code is:

```
12 -> Ryan Carbajal
20 -> Split Mountain
23 -> 23 miles from Cortez
26 -> 14 mi.  W. Sahuarita
33 -> Northeast of Tucson
40 -> Wildland Fire
53 -> Klamath National Forest
56 -> 3 miles East of Tusayan near FR 302 & 2709
61 -> Selway-Bitterroot Wilderness, Nez Perce-Clearwater...
67 -> Page For Emerging Initial Attack Incidents On The...
76 -> 4 miles south of White Wolf Campground and 1 mile...
78 -> Keyesville Area - Black Gulch North
97 -> Peloncilo Mountain Wilderness
98 -> SAN GABRIEL CYN RD/ N RANCH RD
```

After a 20 minutes search in Google and using the indexes of the last output we can build a dictionary with *values* as the name of the states and *keys* as the indexes where the information is going to be inserted:

```
last_states={20:'California', 23:'Colorado', 26:'Arizona',
             33:'Arizona', 53:'Oregon', 56:'Arizona', 61:'Idaho',
             67:'Nevada', 76:'California', 78:'California',
             97:'Arizona', 98:'California'}
```

And now we set the missing states:

```
for key, values in last_states.items():
        wildf_dfv21.at[key, 'Region'] = values
```

There were two locations where we could not pinpoint the name of the states but after a thorough search our efforts paid off:

```
wildf_dfv21.at[12, 'Region'] = 'New Mexico'
wildf_dfv21.at[40, 'Region'] = 'California'
```

After this we drop the columns *Location* and *Incident*:

```
wildf_dfv21.drop(['Location'], axis=1, inplace=True)
wildf_dfv21.drop(['Incident'], axis=1, inplace=True)
```

And finally we save the data frame as a *csv* file:

```
wildf_dfv21.to_csv('blazes/fires_v3.csv', index = False)
```

□