Canadian Crop Yield & Climate Analysis

Functionality for the Program

Functionality for this program is contingent upon several external factors, including your operating system's ability to run Python and the libraries and installation packages listed; see the next section for a list of installation packages and libraries used for this program.

It is important to note that the data cleaning process may take **up to 10 minutes**.

Functionality is also based on the Statistics Canada's API, which we use as the source for the crop yield data. We are not responsible should any errors occur with the API or their website.

Installation Packages

The following installation packages and libraries are required in order for this program to function:

- Statscan (Statistics Canada)
- Sklearn (Scikit-learn)
- Numpy (Numerical Python)
- Matplotlib
- Statsmodels
- Pandas

Reading the Data	
Before you can begin to look for relationships or visualize the data, you need to ***find*** *the data and import it. For this program, we used StatsCan's API[1] to bring crop data yields into our program, and we used a publicly shared dataset hosted on Kaggle[2], compiled from data from Climate Change Canada from one or more weather stations in each Canadian province.*	
Function	**Explanation / Arguments / Returns**

`def redo_columns(df):`
`df['Alberta'] = df[['CALGARY', 'EDMONTON']].mean(axis=1)`
 	
`df['Quebec'] = df[['MONTREAL', 'QUEBEC']].mean(axis=1)`
 	
`df['Ontario'] = df[['OTTAWA', 'TORONTO']].mean(axis=1)`
	• Find means for provinces that have more than one weather station; amalgamate these averages into new columns named for provinces and discard remaining columns

`new_columns = {'MONCTON': 'New Brunswick', 'SASKATOON': Saskatchewan', 'STJOHNS': 'Newfoundland and Labrador', 'VANCOUVER': 'British Columbia', 'WINNIPEG': 'Manitoba', 'HALIFAX': 'Nova Scotia', 'CHARLOTTETOWN': 'Prince Edward Island'}`
 	
`df.rename(columns=new_columns, inplace=True)`
	• Create dictionary to replace weather station names with provinces

`df.drop(['CALGARY','EDMONTON','MONTREAL','OTTAWA','QUEBEC','TORONTO','WHITEHORSE'], axis = 1,inplace=True)`
 	
`return df`
	• Remove columns that provided means
	
	• Return columns with province names

[1] Government of Canada. (2021). Reports and statistics data for Canadian principal field crops. https://aimis-simia.agr.gc.ca/rp/index-eng.cfm?action=pR&r=243&lang=EN
[2] Turner, A. (2019). 80 years of Canadian climate data. *Kaggle.* https://www.kaggle.com/aturner374/eighty-years-of-canadian-climate-data

Possible Errors

As mentioned in the ‘Functionality’ section, functionality for this program is dependent on Statistics Canada’s API. Due to a security breach around 10 December, the Statistics Canada website was unavailable for access. As of 18 December, the website appears to be back up and running, but any similar problems in future will lead to similar errors.

Data Cleaning

Here you’ll tidy the data for consistency and to make it more comfortable to work with.

<table>
<tr><th>Function</th><th>Explanation / Arguments / Returns</th></tr>
<tr><td><pre>def inTemp(row):
  t=prov_ag.iloc[[row.name]]
  y=t['YEAR']
  y=int(y)

p=t['GEO'].astype("string").to_string()
  head, sep, p = p.partition(' ')
  p=p.lstrip()
  test=temp.loc[temp['YEAR'] == y]
  v=test[p]
  return float(v)</pre></td><td>Get a row index of the current row
Find the year associated with the row
Ensure it's an integer
Get province as string (redundancy needed)
Clean extra space created by making the string
Keep removing extra spaces
Get the temperatures for the given year
Select the temp for the given province within selected year
Return temp</td></tr>
<tr><td><pre>def inPrecip(row):
  t=prov_ag.iloc[[row.name]]
  y=t['YEAR']
  y=int(y)</pre></td><td>Similar to inTemp, adds precipitation to agricultural data</td></tr>
</table>

p=t['GEO'].astype("string").to_string()
  head, sep, p = p.partition(' ')
  p=p.lstrip()
  test=precip.loc[precip['YEAR'] == y]
  v=test[p]
  return float(v)	

GUI Creation

Now that all of your data is cleaned, you can begin to design the interface so that the data can be visually understood through an interactive user experience. We opted to use tkinter (tk) as the interface as it is the standard user interface library for Python.

Function	**Explanation / Arguments / Returns**
province_list = sorted(crop_yield["GEO"].unique()) crop_list = sorted(crop_yield["Type_of_crop"].unique()) prov_ag = crop_yield	• Generates the values for the drop down menus and sorts the lists for easier readability
def expected_vector(): tkinter.messagebox.showerror("TypeError", "Expected non-empty vector for x: \nData has no value for province and crop selected")	• Error message generation for bad values (mostly in Newfoundland set)

class MainGUI:
      def __display__(self):
      Graph()
	• Class for the main GUI - presents the user with options to select province, crop type, and climate type to generate a plot, as well as to exit the application

def __init__(self, master): self.master = master	• Generates frame and self-reference for later functions

`Frame1 = Frame(self.master)`
`Frame1.grid()`
`…`
`self.interact()`	

`self.Province = StringVar()`
`self.Province.set("Alberta")`
`self.ProvinceSelect = OptionMenu(master, self.Province, *province_list, command = lambda _: self.getProvince())`
`self.ProvinceSelect.grid(row = 2, column = 2, pady = 5, padx = 5)`
 	
`self.ProvinceSelect["highlighthickness"]=0`
`self.ProvinceSelect.config(bg = "Slategray3")`
`self.ProvLabel = Label(master, text = "Select a province:", fg = "white", bg = def_back,)`
`self.ProvLabel.grid(row = 2, column = 1, padx = 5)`
	• Creates drop-down menu so user can select one of the options from the list taken from above
	• The code for the Crop value and the Climate value are virtually identical
	• This also generates a label to explain the point of the menu to the user

`def interact(self):`
 	
`self.button_1 = Button(self.master , text="Plot", command= lambda : Graph(self.Province.get(), self.CropType.get(), self.ClimateType.get()), bg="papayawhip")`
`self.button_1.grid(row = 5, column = 2, pady = 10, padx = 5)`
 	
`self.button_2 = Button(self.master , text="Quit",`
	• First button calls the Graph class functions and plots the data based on user input from the drop down menus
	• Button that calls the _quit function to exit the program

`command=self._quit, bg="salmon")`
`      self.button_2.grid(row = 6,`
`column = 2, pady = 10, padx = 5)`	

`def _quit():`
`    window.quit()`
`    window.destroy()`
	• Creates a definition for quit
	• Stops the mainloop

`      def getProvince(self):`
`      global Province`
`      ProvinceGet =`
`self.Province.get()`
 	
`      def getCrop(self):`
`      global CropGet`
`      CropGet = self.CropType.get()`
 	
`      def getClimate(self):`
`      global ClimateGet`
`      ClimateGet =`
`self.ClimateType.get()`
	• These 3 functions store the option selected via the drop down menus as a value to be worked with
	• If no option is selected, the default values of "Alberta", "Barley", and "TEMP" will be used

Possible Errors

A user may select a province wherein data was not collected for a particular crop, or the weather station was not yet collecting climate data. In this case, the error message def expected_vector, as outlined above, creates an error message for the user.

Linear Regression Analysis[3]	
At this point, you may choose to have your program analyze significant relationships between the crop data, and the climate or precipitation data. Note that all recommended installations, for analysis and for the full program, can be found in the Installation Packages section.	
Step 1: Testing the existence of a linear relationship between climate change and agricultural crop yield in Canada.	
Function	**Explanation / Arguments / Returns**

`class LinearValidate:`
 	
`  def __init__(self):`
`    self.Harvest_disposition = 'Average yield (kilograms per hectare)'`
`    self.ProvLst = prov_ag[(prov_ag.YEAR  == 2018) &`
 	
`(prov_ag.Type_of_crop  == DEFAULT_CROP) &`
 	
`(prov_ag.Harvest_disposition == self.Harvest_disposition)].GEO.values`
`    self.ProvRecord = []`
`    self.CropRecord = []`
`    self.FTestRecord = []`
`    self.validate(prov_ag)`
	• Class to summarize if there is a linear correlation between each pair of variables (temp, precip) and (yield)
	• Initiate class variables to store a list of all provinces, a list of crops grown in a province and a list of results documenting the applicability for a linear model

`def ProvCropType(self, Df_ag, Prov):` `    return prov_ag[(prov_ag.GEO ==`	• Return all the crop types grown in a particular province

[3] Note that while this is not part of the program, we included code for those interested in looking into significant relationships among the data.

Prov) & (prov_ag.YEAR == DEFAULT_YEAR) & (prov_ag.Harvest_disposition == self.Harvest_disposition)].Type_of_crop.values	

def validate(self, Df_ag):
    for ProvName in self.ProvLst:
      ThisProvCrop = self.ProvCropType(Df_ag, ProvName)
      for CropName in ThisProvCrop:
 	
        ThisCropDf = prov_ag[(prov_ag.GEO == ProvName) & (prov_ag.Type_of_crop == CropName) & (prov_ag.Harvest_disposition == self.Harvest_disposition)].dropna()
 	
        X = ThisCropDf.loc[:, ['TEMP', 'PRECIP']].values
        y = ThisCropDf.loc[:, ['VALUE']].values
 	
        if (len(X) == len(y)) and (len(X) != 0):
 	
          X = sm.add_constant(X)
          model = sm.OLS(y,X).fit()
 	
          if (model.pvalues[0] < SIGNIFICANCE_LEVEL) and \\
          (model.pvalues[1] < SIGNIFICANCE_LEVEL) and \\
          (model.pvalues[2] < SIGNIFICANCE_LEVEL):
	• Generate a third column indicating the applicability of a linear model to summarize the relation between crop yield and (temperature/precipitation change)
	• Do an f test to validate a linear model's fit for this crop's data in 80 years
	• Extract the 80-year historical data for ThisProvCrop
	
	• Extract dependent variables y and the independent variables X
	
	• Fit a linear model with (X, y)
	
	• Run an f-test for the validity of this model

<pre>   f_test = 'Yes'
          else:
            f_test = 'No'
 	
self.ProvRecord.append(ProvName)
self.CropRecord.append(CropName)
self.FTestRecord.append(f_test)
 	
    LinearRegressionTest =
{'Province_name': self.ProvRecord,
'Crop_name': self.CropRecord,
'Existence_of_linear_relation':
self.FTestRecord}
 	
    self.result =
pd.DataFrame(LinearRegressionTest,
columns = ['Province_name',
'Crop_name',Existence_of_linear_rel
ation'])</pre>
	• Push the result of the f test to self.record
	
	• After the iteration, create the new dataframe

<pre> def __print__(self): display(self.result)</pre>	• Display all the crops grown in each province and if there is a linear relation between climate change and crop yield

<pre>def __filter__(self):
    self.filtered =
self.result[self.result.Existence_o
f_linear_relation == 'Yes']
    display(self.filtered)</pre>
	• Display only the ones where a linear correlation exists between the yield of this crop grown in that province and the change in temperature as well as precipitation in the past 80 years

Step 2: Visualize the comparison between:

A: trends in temperature change,

B: trends in precipitation change,

C: change in the yield of a particular crop species in a Canadian province in the past 80 years.

The class is designed based on the assumption that the user has been provided with entry boxes/a drop-down list including:

A: province

B: a valid CropName (this might be different from province to province; see 'Possible Errors' at the end of this section)

C: Either 'Temperature' or 'Precipitation'

Function	Explanation / Arguments / Returns

`class Graph:`
 	
`   def __init__(self, Prov, CropTp, TempORPrecip):`
`    self.Prov = Prov`
`    self.CropTp = CropTp`
`    self.TempORPrecip = TempORPrecip`
`    self.__setData__()`
`    self.__display__()`
	• The class Graph displays the visualization of B and C's trend in the past 80 years
	• Set the inputs given by users as class variables, so that the province names, crop names, and environmental data can be used to create the plots

`  def __setData__(self):` `    self.X = (prov_ag[(prov_ag.GEO == self.Prov) & (prov_ag.Type_of_crop == self.CropTp) & (prov_ag.Harvest_disposition == 'Average yield (kilograms per hectare)')          ].dropna()["YEAR"]).values`	• Extract all values which correspond to the provincial names, crop names, temperature values or precipitation value queries by users, and set the historical values of crop yield as an array of dependent variables and the temp/ precip values as an array of dependent variables

<pre>   self.production =
(prov_ag[(prov_ag.GEO == self.Prov)
& (prov_ag.Type_of_crop ==
self.CropTp) &
(prov_ag.Harvest_disposition ==
'Average yield (kilograms per
hectare)')   ].dropna(
)["VALUE"]).values
    if self.TempORPrecip ==
'Temperature':
      self.key = 'TEMP'
    else:
      self.key = 'PRECIP'
    self.independentVAR =
(prov_ag[(prov_ag.GEO == self.Prov)
&
(prov_ag.Type_of_crop ==
self.CropTp) &
(prov_ag.Harvest_disposition ==
'Average yield (kilograms per
hectare)')   ].dropna(
)[self.key]).values
 	
  def __display__(self):
    self.max_temp =
max((prov_ag["TEMP"].dropna()).valu
es)
    self.min_temp =
min((prov_ag["TEMP"].dropna()).valu
es)
    self.max_precip =
max((prov_ag["PRECIP"].dropna()).va
lues)
    self.max_yield =
max((prov_ag[(prov_ag.Type_of_crop
== self.CropTp) &
(prov_ag.Harvest_disposition ==
‘Average yield (kilograms per
hectare)')]["VALUE"]   .dro</pre>
	• Display the scatter plots and the linear regression lines of dependent and independent variables and find maximum and minimums for setting the scale of axis displayed

<pre>pna()).values)
 	
    fig, ax1 = plt.subplots()
    ax2 = ax1.twinx()
    ax1.plot(self.X,
self.production, 'o', color =
'lightcoral', label = 'Average
Yield')
    ax1.set_xlabel('Year')
    ax1.set_ylabel('kilograms per
hectare')</pre>
	• Declare a subplot
	• Set the x-axis and left y-axis of the plot

<pre>   if self.key == 'TEMP':
      ax2.plot(self.X,
self.independentVAR, 'o', color =
'peachpuff', label = 'Temperature')
      ax2.set_ylim(2, 15)
      ax2.set_ylabel('Celsius')
    else:
      ax2.plot(self.X,
self.independentVAR, 'o', color =
'lightblue', label =
'Precipitation')
      ax2.set_ylim(0, 4)
      ax2.set_ylabel('100mm')</pre>
	• Set the right y-axis of the plot

<pre> fig.legend(loc='upper left', bbox_to_anchor=(0.12, 0.9))</pre>	• Set and display a legend for the plot

<pre>   m, b = np.polyfit(self.X,
self.production, 1)
    m1, b1 = np.polyfit(self.X,
self.independentVAR, 1)
 	
    ax1.plot(self.X, m*self.X+b,
color='lightcoral')
    if self.key == "TEMP":
      ax2.plot(self.X,
m1*self.X+b1, color='peachpuff')</pre>
	• Set and display the linear regression lines of the independent and dependent variables

<pre> else: ax2.plot(self.X, m1*self.X+b1, color='lightblue')</pre>	
Possible Errors	
Because the collected data fluctuated from province to province, this will affect the analysis and the visualization of the data.	