Part 2 Regression

Regression:

- supervised learning
- the target variable is numeric and continuous

8 Predicting numeric values: regression

8.1 Finding best-fit lines with linear regression

- **Pros:** Easy to interpret results, computationally inexpensive
- **Cons:** Poorly models nonlinear data
- **Works with:** Numeric values, nominal values

Regression: the process of finding regression weights → regression equation + regression weights

- Linear regression

  The output is the summation of inputs multiplied by some constants

- Nonlinear regression

  The output is a function of inputs multiplied together

General approach to regression:

1. **Collect:** Any method.
2. **Prepare:** We'll need numeric values for regression. Nominal values should be mapped to binary values.
3. **Analyze:** It's helpful to visualized 2D plots. Also, we can visualize the regression weights if we apply shrinkage methods.
4. **Train:** Find the regression weights.
5. **Test:** We can measure the R2, or correlation of the predicted value and data, to measure the success of our models.
6. **Use:** With regression, we can forecast a numeric value for a number of inputs. This is an improvement over classification because we're predicting a continuous value rather than a discrete category.

Input data: the matrix $\mathbf{X}$ and the regression weights vector w

The way to find w: minimize the error → using the squared error

$$\sum_{i=1}^{m}(y_i - x_i^T w)^2$$
$$or$$
$$(y - \mathbf{X}w)^T(y - \mathbf{X}w)$$

Take the derivative of $(y - \mathbf{X}w)^T(y - \mathbf{X}w)$ and get $\mathbf{X}^T(y - \mathbf{X}w)$, which should be set to zero and solve:

$$\hat{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y$$

Note: $\mathbf{X}^{-1}$ may not exist. Check in advance.

The method above is known as **OLS**, which stands for "ordinary least squares".

Listing 8.1 **Standard regression function and data importing functions: `loadDataSet()` & `standRegres()`**

```
from numpy import *

def loadDataSet(fileName):
    numFeat = len(open(fileNmae).readline().split('\t')) - 1
    dataMat = []
    labelMat = []
    fr = open(fileName)
    for line in fr.readlines():
        lineArr = []
        curLine = line.strip().split('\t')
        for i in range(numFeat):
            lineArr.append(float(curLine[i]))
        dataMat.append(lineArr)
        labelMat.append(float(curLine[-1]))
    return dataMat, labelMat

def standRegres(xArr, yArr):
    xMat = mat(xArr)
    yMat = mat(yArr).T
    xTx = xMat.T * xMat
    if linalg.det(xTx) == 0.0:
        print("This matrix is singular, cannot do inverse")
        return
    ws = xTx.I * (xMat.T * yMat)
    return ws
```

- `if linalg.det(xTx) == 0.0:`

  NumPy has a linear algebra library called linalg, which has a number of useful functions.

  `linalg.det()`: Compute the determinate.

- `ws = xTx.I * (xMat.T * yMat)`

  Using `ws = linalg.solve(xTx.I, xMat.T*yMat)` is also proper.

The NumPy command to generate the correlation coefficients: `corrcoef(x, y)`

8.2 Locally weighted linear regression

Linear regression tends to unferfit the data.

→ To reduce the mean-squared error by adding some bias into the estimator

Locally weightedlinear regression (LWLR):

1. Give a weight to data points near the data point of interest
2. Compute a least-squares regression

$$\hat{w} = (\mathbf{X}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^T \mathbf{W} y$$

$\mathbf{W}$ is the matrix used to weight the data points.

LWLR uses a kernel to weight nearby points more heavily than other points. The most common kernel to use is a Gaussian which assigns a weight given by

$$w(i,i) = \exp\left(\frac{|x^{(i)} - x|}{-2k^2}\right)$$

This builds the weight matrix $\mathbf{W}$, which has only diagonal elements. The closer the data point x is to the other points, the larger $w(i,i)$ will be. There also is a user-defined constant k that will determine how much to weight nearby points.

Listing 8.2 **Locally weighted linear regression function: `lwlr()` & `lwlrTest()`**

See Jupyter.

The parameter `k` should be adjusted properly to avoid underfitting as well as overfitting.

→ Better trying several different values of `k` and making a mutual comparison

→ Sometimes LWLR *may not* outperform the simple linear regression

8.3 Example: predicting the age of an abalone

See Jupyter.

8.4 Shrinking coefficients to understand our data

Goal: To deal with the incurring problems led by more features than data points

8.4.1 Ridge regression

Ridge regression adds an additional matrix $\lambda\mathbf{I}$ to the matrix $\mathbf{X}^T\mathbf{X}$ so that it's non-singular, and we can take the inverse of the whole thing: $\mathbf{X}^T\mathbf{X} + \lambda\mathrm{I}$.

$$\hat{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathrm{I})^{-1}\mathbf{X}^T y$$

Shrinkage: To decrease unimportant parameters by imposing a maximum value on the sum of all `w`s with the λ value.

We choose λ to minimize prediction error.

Listing 8.3 **Ridge regression: `ridgeRegres()` & `ridgeTest()`**

See Jupyter.

Other shrinkage methods: the lasso, LAR, PCA regression, and subset selection

8.4.2 The lasso

The equation for ridge regression is the same as the regular least-squares regression and imposing the following constraint:

$$\sum_{k=1}^{n} w_k^2 \leq \lambda$$

$\rightarrow$ The sum of the squares of all the weights has to be less than or equal to λ.

When two or more of the features are correlated, we may have a very large positive weight and a very large negative weight using regular least-squares regression.

$\rightarrow$ Can be avoided by ridge regression thanks to the constraint above

The **lasso** imposes a different constraint on the weights:

$$\sum_{k=1}^{n} |w_k| \leq \lambda$$

When taking the absolute value instead of the square, some of the weights are forced to be exactly 0 if λ is small enough.

How to solve the inequality above:

- Apply a quadratic programming algorithm
- Apply forward stagewise regression (approximate)

8.4.3 Forward stagewise regression

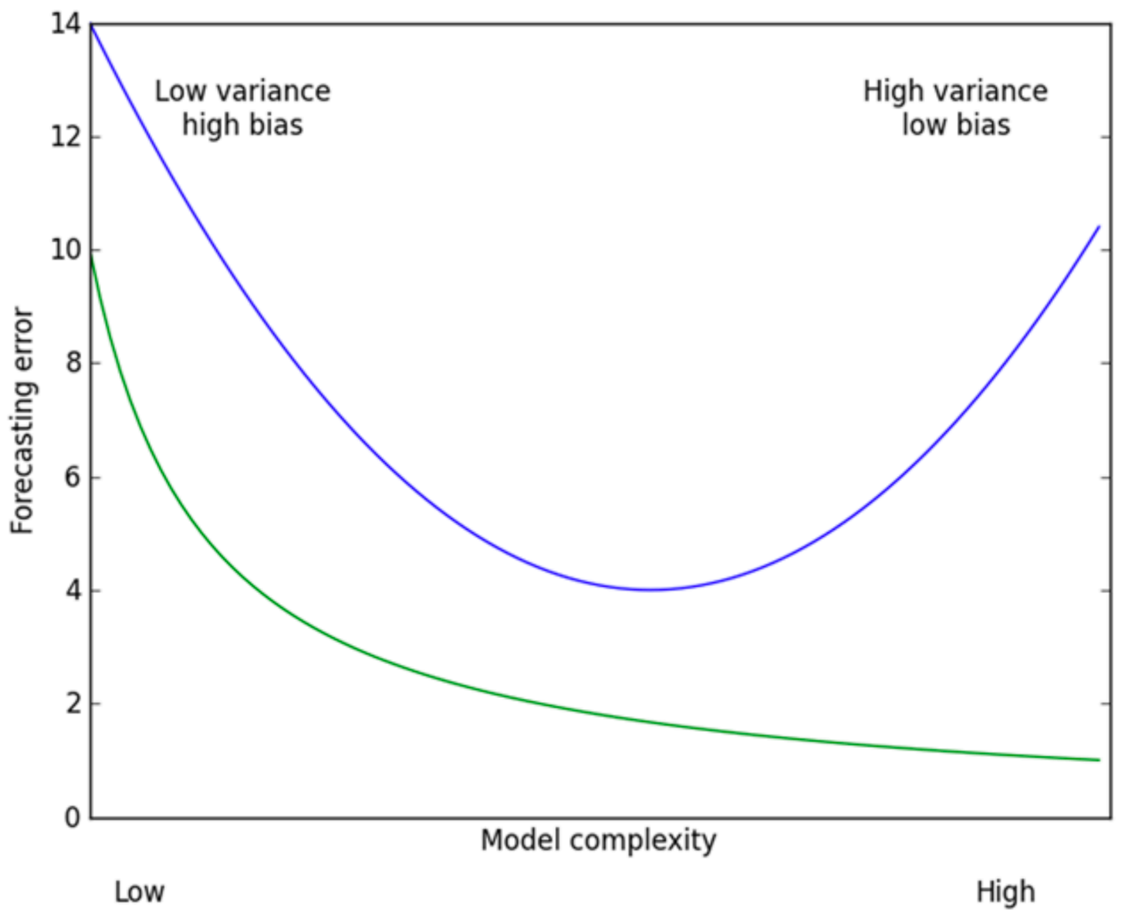
Pseudocode for `stageWise()`:

```
Regularize the data to have 0 mean and unit variance
For every iteration:
    set lowestError to $+\infty$
    For every feature:
        For increasing and decreasing:
            Change one coefficient to get a new W
            Calculate the Error with new W
            If the Error is lower than lowestError: set Wbest to the current W
        Update set W to Wbest
```

Listing 8.4 **Forward stagewise linear regression: `stageWise()`**

See Jupyter.

8.5 The bias/variance tradeoff



The blue curve is the test error and the green curve is the training error. We should adjust the model complexity where the test error is at a minimum.

Errors can be viewed as the sum of:

- bias

- error
- random noise

8.6 Example: forecasting the price of LEGO sets

Example: using regression to predict the price of a LEGO set

1. **Collect:** Collect from Google Shopping API.
2. **Prepare:** Extract price data from the returned JSON.
3. **Analyze:** Visually inspect the data.
4. **Train:** We'll build different models with stagewise linear regression and straight-forward linear regression.
5. **Test:** We'll use cross validation to test the different models to see which one performs the best.
6. **Use:** The resulting model will be the object of this exercise.

8.6.1 Collect: using the Google shopping API

Listing 8.5 **Shopping information retrieval function: `searchForSet()` & `setDataCollect()`**

```
from time import sleep
import json
import urllib.request   # 'urllib2' is not available in Python3

def searchForSet(retX, retY, setNum, yr, numPce, origPrc):
    sleep(10)
    myAPIstr = 'get%20from%20code.google.com'
    searchURL = 'https://www.googleapis.com/shopping/search/v1/public/products?
key=%s&country=US&q=lego+%d&alt=json' % (myAPIstr, setNum)
    pg = urllib.request.urlopen(searchURL)
    retDict = json.loads(pg.read())
    for i in range(len(retDict['items'])):
        try:
            currItem = retDict['items'][i]
            if currItem['product']['condition'] == 'new':
                newFlag = 1
            else:
                newFlag = 0
            listOfInv = currItem['product']['inventories']
            for item in listOfInv:
                sellingPrice = item['price']
                if sellingPrice > origPrc * 0.5:
                    print("%d\t%d\t%d\t%f\t%f" % (yr, numPce, newFlag, origPrc,
sellingPrice))
                    retX.append([yr, numPce, newFlag, origPrc])
                    retY.append(sellingPrice)
        except:
            print('problem with item %d' % i)

def setDataCollect(retX, retY):
    searchForSet(retX, retY, 8288, 2006, 800, 49.99)
    searchForSet(retX, retY, 10030, 2002, 3096, 269.99)
    searchForSet(retX, retY, 10179, 2007, 5195, 499.99)
    searchForSet(retX, retY, 10181, 2007, 3428, 199.99)
    searchForSet(retX, retY, 10189, 2008, 5922, 299.99)
    searchForSet(retX, retY, 10196, 2009, 3263, 249.99)
```

- `sleep(10)`

  `time.sleep(secs)` → Suspend execution of the calling thread for the given number of seconds.

  Sleep for 10 seconds at first as a precaution to prevent making too many API calls too quickly.

- `pg = urllib.request.urlopen(searchURL)`

  `urllib.request.urlopen(url, data=None, [timeout, ]*, cafile=None, capath=None, cadefault=False, context=None)` → Open the URL `url`, which can be either a string or a `Request` object.

- `retDict = json.loads(pg.read())`

  `json.loads(s, *, encoding=None, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, *kw)` → Deserialize s (a `str`, `bytes` or `bytearray` instance containing a JSON document) to a Python object using this conversion table. The other arguments have the same meaning as in `load()`, except encoding which is ignored and deprecated.

8.6.2 Train: building a model

Listing 8.6 **Cross-validation testing with ridge regression: `crossValidation()`**

```
def crossValidation(xArr, yArr, numVal = 10):
    m = len(yArr)
    indexList = list(range(m))
    errorMat = zeros((numVal, 30))
    for i in range(numVal):
        trainX = []
        trainY = []
        testX = []
        testY = []    # Create training and test containers
        random.shuffle(indexList)
        for j in range(m):
            if j < m * 0.9:
                trainX.append(xArr[indexList[j]])
                trainY.append(yArr[indexList[j]])
            else:
                testX.append(xArr[indexList[j]])
                testY.append(yArr[indexList[j]])   # Split data into test and
training sets
        wMat = ridgeTest(trainX, trainY)
        for k in range(30):
            matTestX = mat(testX)
            matTrainX = mat(trainX)
            meanTrain = mean(matTrainX, 0)
            varTrain = var(matTrainX, 0)   # Regularize test with training params
            matTestX = (matTestX - meanTrain) / varTrain
            yEst = matTestX * mat(wMat[k,:]).T + mean(trainY)
            errorMat[i,k] = rssError(yEst.T.A, array(testY))
    meanErrors = mean(errorMat, 0)
    minMean = float(min(meanErrors))
    bestWeights = wMat[nonzero(meanErrors == minMean)]
    xMat = mat(xArr)
    yMat = mat(yArr).T
    meanX = mean(xMat, 0)
    varX = var(xMat, 0)
    unReg = bestWeights / varX
    print("the best model from Ridge Regression is:\n", unReg)
    print("with constant term: ", -1*sum(multiply(meanX, unReg)) + mean(yMat))
# Undo regularization
```

- **Inputs:**
  - *xArr, yArr:* lists of the X and Y values of a dataset
  - *numVal:* the number of cross validations to run
- `random.shuffle(indexList)`

  `random.shuffle(x[, random])` → Shuffle the sequence `x` in place.
- `for k in range(30):`

  Loop over all 30 sets of weights and test them using the test data.
- `matTestX = (matTestX - meanTrain) / vatTrain`

  Ridge regression assumes that the data has been normalized, co the test data should be normalized with the same parameters used to normalize the training data.

8.7 Summary

9 Tree-based Regression

9.1 Locally modeling complex data

- **Pros:** Fits complex, nonlinear data
- **Cons:** Difficult to interpret results
- **Works with:** Numeric values, nominal values

Binary split: Continuous values greater than the desired value go on the left side of the tree and all the other values go on the rights side.

CART (Classification And Regression Trees) is a well-known and well-documented tree-building algorithm that makes binary splits and handles continuous variables.

Regression trees are similar to trees used for classification but with the leaves representing a numeric value rather than a discrete one.

General approach to tree-based regression:

1. **Collect:** Any method.
2. **Prepare:** Numeric values are needed. If you have nominal values, it's a good idea to map them into binary values.
3. **Analyze:** We'll visualize the data in two-dimensional plots and generate trees as dictionaries.
4. **Train:** The majority of the time will be spent building trees with models at the leaf nodes.
5. **Test:** We'll use the R^2 value with test data to determine the quality of our models.
6. **Use:** We'll use our trees to make forecasts. We can do almost anything with these results.

9.2 Build trees with continuous and discrete features

A **dictionary** will be used to store the different types of data making up the tree. The dictionary will have the following four items:

- *Feature:* A symbol representing the feature split on for this tree
- *Value:* The value of the feature used to split
- *Right:* The right subtree; this could also be a single value if the algorithm decides we don't need another split.
- *Left:* The left suntree similar to the right subtree.

Two types of trees:

- Regression tree → containing a single value for each leaf of the tree
- Model tree → having a linear equation at each leaf node

Pseudocode for `createTree()`:

```
Find the best feature to split on:
  If we can't split the data, this node becomea a leaf node
  Make a binary split of the data
  Call createTree() on the right split of the data
  Call createTree() on the left split of the data
```

Listing 9.1 **CART tree-building code: `loadDataSet()` & `binSplitDataSet()` & `createTree()`**

See Jupyter.

9.3 Using CART for regression

9.3.1 Building the tree

Pseudocode for `chooseBestSplit()`:

```
For every feature:
    For every unique value:
        Split the dataset to two
        Measure the error of the two splits
        If the error is less than bestError → set bestSplit to this split and update bestError
Return bestSplit feature and threshold
```

Listing 9.2 **Regression tree split function: `regLeaf()` & `regErr()` & `chooseBestSplit()`**

See Jupyter.

9.3.2 Executing the code

See Jupyter.

9.4 Tree pruning

Pruning: the procedure of reducing the complexity if a decision tree to avoid overfitting

- Prepruning → Prune the tree as it's being built

  More effective

- Postpruning → Prune the tree after it's built

9.4.1 Prepruning

The trees built through `createTree()` above are sensitive to the settings `tolS` and `tolN` since `tolS` is sensitive to the magnitude of the errors.

9.4.2 Postpruning

The postpruning methos will first split our data into a test set and a training set.

Pseudocode for `prune()`:

```
Split the test data for the given tree:
    If the either split is a tree:
        Call prune on that split
    Calculate the error associated with merging two leaf nodes
```

Calculate the error without merging
If merging results in lower error then merge the leaf nodes

Listing 9.3 **Regression tree-pruning functions: `isTree()` & `getMean()` & `prune()`**

```
def isTree(obj):
    return (type(obj).__name__ == 'dict')

def getMean(tree):
    if isTree(tree['right']):
        tree['right'] = getMean(tree['right'])
    if isTree(tree['left']):
        tree['left'] = getMean(tree['left'])
    return (tree['left'] + tree['right']) / 2.0

def prune(tree, testData):
    if shape(testData)[0] == 0:
        return getMean(tree)   # Collapse tree if no test data
    if isTree(tree['right']) or isTree(tree['left']):
        lSet, rSet = binSplitDataSet(testData, tree['spInd'], tree['spVal'])
    if isTree(tree['left']):
        tree['left'] = prune(tree['left'], lSet)
    if isTree(tree['right']):
        tree['right'] = prune(tree['right'], rSet)
    if not isTree(tree['left']) and not isTree(tree['right']):
        lSet, rSet = binSplitDataSet(testData, tree['spInd'], tree['spVal'])
        errorNoMerge = sum(power(lSet[:,-1] - tree(['left'], 2))) +
sum(power(rSet[:,-1] - tree(['right'], 2)))
        treeMean = (tree['left'] + tree['right']) / 2.0
        errorMerge = sum(power(testData[:,-1] - treeMean, 2))
        if errorMerge < errorNoMerge:
            print("merging")
            return treeMean
        else:
            return tree
    else:
        return tree
```

- **Functions:**
  - `isTree()`: Test if a variable is a tree and return a Boolean type.
  - `getMean()`: A recursive function that descends a tree until it hits only leaf nodes. When it finds two leaf nodes, it takes the average of these two nodes.
  - `prune()`: Prune th tree.

```
if shape(testData)[0] == 0:
  return getMean(tree)   # Collapse tree if no test data
```

When some instant where the test data doesn't contain the value in the same range as the original dataset, this branch will be assumed as overfitted.

Postpruning is not as effective as prepruning.

9.5 Model trees

Piecewise linear model: A model that consists of multiple linear segments

→ Easier to interpret and more accurate

Listing 9.4 **Leaf-generation function for model trees: `linearSolve()` & `modelLeaf()` & `modelErr()`**

See Jupyter.

9.6 Example: comparing tree methods to standard regression

Listing 9.5 **Code to create a forecast with tree-based regression: `regTreeEval()` & `modelTreeEval()` & `treeForeCast()` & `createForeCast()`**

```
def regTreeEval(model, inDat):
    return float(model)

def modelTreeEval(model, inDat):
    n = shape(inDat)[1]
    X = mat(ones((1, n+1)))
    X[:,1:(n+1)] = inDat
    return float(X * model)

def treeForeCast(tree, inData, modelEval = regTreeEval):
    if not isTree(tree):
        return modelEval(model, inData)
    if inData[tree['spInd']] > tree['spVal']:
        if isTree(tree['left']):
            return treeForeCast(tree['left'], inData, modelEval)
        else:
            return modelEval(tree['left'], inData)
    else:
        if isTree(tree['right']):
            return treeForeCast(tree['right'], inData, modelEval)
        else:
            return modelEval(tree['right'], inData)
```

```
def createForeCast(tree, testData, modelEval = regTreeEval):
    m = len(testData)
    yHat = mat(zeros((m,1)))    # Mistake corrected
    for i in range(m):
        yHat[i,0] = treeForeCast(tree, mat(testData[i]), modelEval)
    return yHat
```

- **Functions:**
    - `regTreeEval()`: The function used to evaluate a regression tree leaf node which has two inputs though only one is used.
    - `modelTreeEval()`: The function used to evaluate a model tree node.
    - `treeForeCast()`: The function takes a single data point or row vector and will return a single floating-point value giving one forecast for one data point for a given tree. It follows the tree based on the input data until a leaf node is hit.
        - `modelEval`: A reference to a function used to evaluate the data at a leaf node whose default value is `regTreeEval`.

9.7 Using Tkinter to create a GUI in Python

GUI (graphical user interface): one method to help present data and offer a way to interact with data

Example: building a GUI to tune a regression tree

1. **Collect:** Text file provided.
2. **Prepare:** We need to parse the file with Python, and get numeric values.
3. **Analyze:** We'll build a GUI with tkinter to display the model and the data.
4. **Train:** We'll train a regression tree and a model tree and display the models with the data.
5. **Test:** No testing will be done.
6. **Use:** The GUI will allow people to play with different settings for prepruning and to choose different types of models to use.

9.7.1 Building a GUI in tkinter

```
from tkinter import *
root = Tk()
myLabel = Label(root, text = "Hello World")
myLabel.grid()
root.mainloop()
```

- `root = Tk()`

`tkinter.Tk(screenName=None, baseName=None, className='Tk', useTk=1)` → The `Tk` class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an application. Each instance has its own associated Tcl interpreter.

- `myLabel = Label(root, text = "Hello World")`

  A GUI in tkinter is made up of widgets, which are things like text boxes, buttons, labels, and check buttons. `myLabel` is the only widget in this example.

- `myLabel.grid()`

  When calling the `.grid()` method, the geometry manager is told where to put `myLabel`. `grid` is one of the geometry managers that puts widgets in a two-dimensional table able to specify the row and column of each widget with a default row 0 and column 0.

- `root.mainloop()`

  This command kicks off the event loop, which handles mouse clicks, keystrokes, and redrawing, among other things.

Listing 9.6 **Tkinter widgets used to build tree explorer GUI: `reDraw()` & `drawNewTree()`**

```
def reDraw(tolS, tolN):
    pass

def drawNewTree():
    pass

root = Tk()
Label(root, text = "Plot Place Holder").grid(row = 0, columnspan = 3)
Label(root, text = "tolN").grid(row = 1, column = 0)
tolNentry = Entry(root)
tolNentry.grid(row = 1, column = 1)
tolNentry.insert(0, '10')
Label(root, text = "tolS").grid(row = 2, column = 0)
tolSentry = Entry(root)
tolSentry.grid(row = 2, column = 1)
tolSentry.insert(0, '1.0')
Button(root, text = "ReDraw", command = drawNewTree).grid(row = 1, column = 2,
rowspan = 3)
chkBtnVar = IntVar()
chkBtn = Checkbutton(root, text = "Model Tree", variable = chkBtnVar)
chkBtn.grid(row = 3, column = 0, columnspan = 2)
reDraw.rawDat = mat(loadDataSet('/Users/duoduo/Desktop/Data/Machine
Learning/Machine Learning in Action/MLA_SourceCode/Ch09/sine.txt'))
reDraw.testDat = arange(min(reDraw.rawDat[:,0]), max(reDraw.rawDat[:,0]), 0.01)
reDraw(1.0, 10)
root.mainloop()
```

- `Label(root, text = "Plot Place Holder").grid(row = 0, columnspan = 3)`

  `columnspan` and `rowspan` can allow a widget to span more than one row or column.

- `tolNentry = Entry(root)`

  `Entry()` → `Entry` widget is a text box where a single line of text can be entered.

  ```
  chkBtnVar = IntVar()
  chkBtn = Checkbutton(root, text = "Model Tree", variable = chkBtnVar)
  ```

  `Checkbutton` and `IntVar` are self-explanatory. `IntVar` is created to read the state of `Checkbutton`.

9.7.2 Interfacing Matplotlib and tkinter

- Matplotlib
  - Frontend: the user-facing code
  - Backend: the interface between plots and many other different applications

    → Using TkAgg

Listing 9.7 **Code for Integrating Matplotlib and tkinter: `reDraw()` & `getInputs()` & `drawNewTree()`**

See Jupyter.

9.8 Summary