

Homework Assignment 4: Text Analysis

For this assignment, you are to write a program to perform some simple text analysis. This analysis will compare pairs of text files using a *difference measure* that tends to be smaller when the two texts are written by the same author.

User Requirements

The user is doing literature research, specifically concerned with identifying the authors of certain works. One technique involves computing the frequencies at which commonly-used words occur. These frequencies are then combined into a difference measure. The user needs a program to read all of the text files in a given folder and produce this difference measure for each of these files when compared to a selected file. Furthermore, the user needs the ability to select the number of words to be used in this computation - anywhere from 1 to 100 words. In what follows, we will define this difference measure.

We first need to define precisely what we consider to be a word. For the purpose of this program, a *word* is any maximal sequence of English letters. Furthermore, for the sake of consistency, all letters will be converted to lower-case. Suppose for example, that an input file consists of the following paragraph from Lewis Carroll's *The Hunting of the Snark*:

For instance, take the two words "fuming" and "furious." Make up your mind that you will say both words, but leave it unsettled which you will first. Now open your mouth and speak. If your thoughts incline ever so little towards "fuming," you will say "fuming-furious;" if they turn, by even a hair's breadth, towards "furious," you will say "furious-fuming;" but if you have the rarest of gifts, a perfectly balanced mind, you will say "frumious."

In this paragraph, the word "if" occurs three times. Even though one of these occurrences begins with an upper-case "I", we count it the same as the other two occurrences. Also, the word "s" occurs once, as the apostrophe in "hair's" splits this word into the words "hair" and "s" by our definition. Likewise, the word "hair's" does not occur, as it isn't a valid word by our definition, and "s" occurs only once, as the other occurrences of this letter can be extended to longer words.

The difference measure is based on the following:

- The *frequency* of a word in a text is the number of occurrences of this word in the text, divided by the total number of words in the text.
- The *minimum frequency* of a word is the minimum of its frequencies in the two texts.

For a given positive integer n , the difference measure uses the n words with the highest minimum frequencies (or all the words with nonzero minimum frequencies if there are fewer than n words with nonzero minimum frequencies). Suppose the frequencies of these words within the first text are x_1, \dots, x_k , and that their frequencies within the second text are y_1, \dots, y_k , where k is the number of words we are using ($k = n$, if possible). If k is at least 1, the difference measure is then given by:

$$d = \sqrt{(x_1 - y_1)^2 + \dots + (x_k - y_k)^2}$$

This measure gives the straight-line distance between the two k -dimensional points (x_1, \dots, x_k) and (y_1, \dots, y_k) . If k is 0, we define the difference measure to be ∞ . If the difference measure is finite, it will always be less than $\sqrt{2}$, which is approximately 1.414 (we prove this fact at the end of this document for those who are interested).

Starting the Assignment

Create a GitHub repository using [this URL](#), and clone it to your local machine. This repository contains a new Windows Forms Application with the following files added:

- **ListViewColumnSorter.cs** from Microsoft's ["Use Visual C# to sort a ListView control by using a column"](#), edited slightly.
- **MinPriorityQueue.cs** from the model solution to Lab Assignment 24. The namespace has been changed to match the namespace of the rest of the project.
- **LeftistTree.cs** from the model solution to Lab Assignment 24. The namespace has been changed, and the code from **LeftistTree.ITree.cs** has been merged into this file in order to cause the class diagram to draw properly.
- **KansasStateUniversity.TreeViewer2.dll**, which is needed by the two files above.

In particular, note that the GUI you are given is a blank form.

User Interface

The [demo video](#) shows the desired behavior of your finished program. Here, we will first summarize the GUI design with some instructions on how you will need to set this up. We will then summarize how the finished program should respond to various user actions.

Note: If at the top of your Design window there is a warning containing a link, "Restart Visual Studio with 100% scaling", click this link before editing the GUI. This will help to ensure that the running program appears the same as what is shown in the Design window.

Your finished program should display a GUI resembling the following:

Text Analyzer

Open Selected File:

All Files Selected

Number of Words Used: 25

File Name	Vocabulary	Difference

Before you begin constructing this GUI, if your Design window has a message in a yellow box at the top with a link, "Restart Visual Studio with 100% scaling", click this link. This will help to ensure that the layout of the running GUI matches the layout in the Design window. Then it will make sense to change to **Font** property of the form to a 12-point font; otherwise, a couple of the characters that need to be displayed will be too small to be seen clearly. Changing the font of the form will change the font of most of the controls you add to the form.

At the top is a **ToolStrip** containing a button, a separator, a label, and an empty text box. The text box should be made read-only (see its **ReadOnly** property). Unfortunately, if you changed the font of the form, this change won't propagate to the **ToolStrip**, but you can change the **Font** property of both the **ToolStrip** and its text box.

Below the **ToolStrip** is a **TabControl**. This control should be docked in the center of the form. By default, when it is added to the form, it will contain two tabs, which is what you will need. The first of these tabs can be accessed by clicking on the area below the two tabs. This control is a **TabPage**. The tab will display the contents of the **TabPage's Text** property.

At the top of the first **TabPage** are a label and a **NumericUpDown**. The range of the **NumericUpDown** (see its **Minimum** and **Maximum** properties) should be 1 to 100, and its initial value should be 25. Clicking the buttons on its right should increase or decrease the value by 5 (see its **Increment** property). The value in the **TextBox** should be aligned to the right (see its **TextAlign** property).

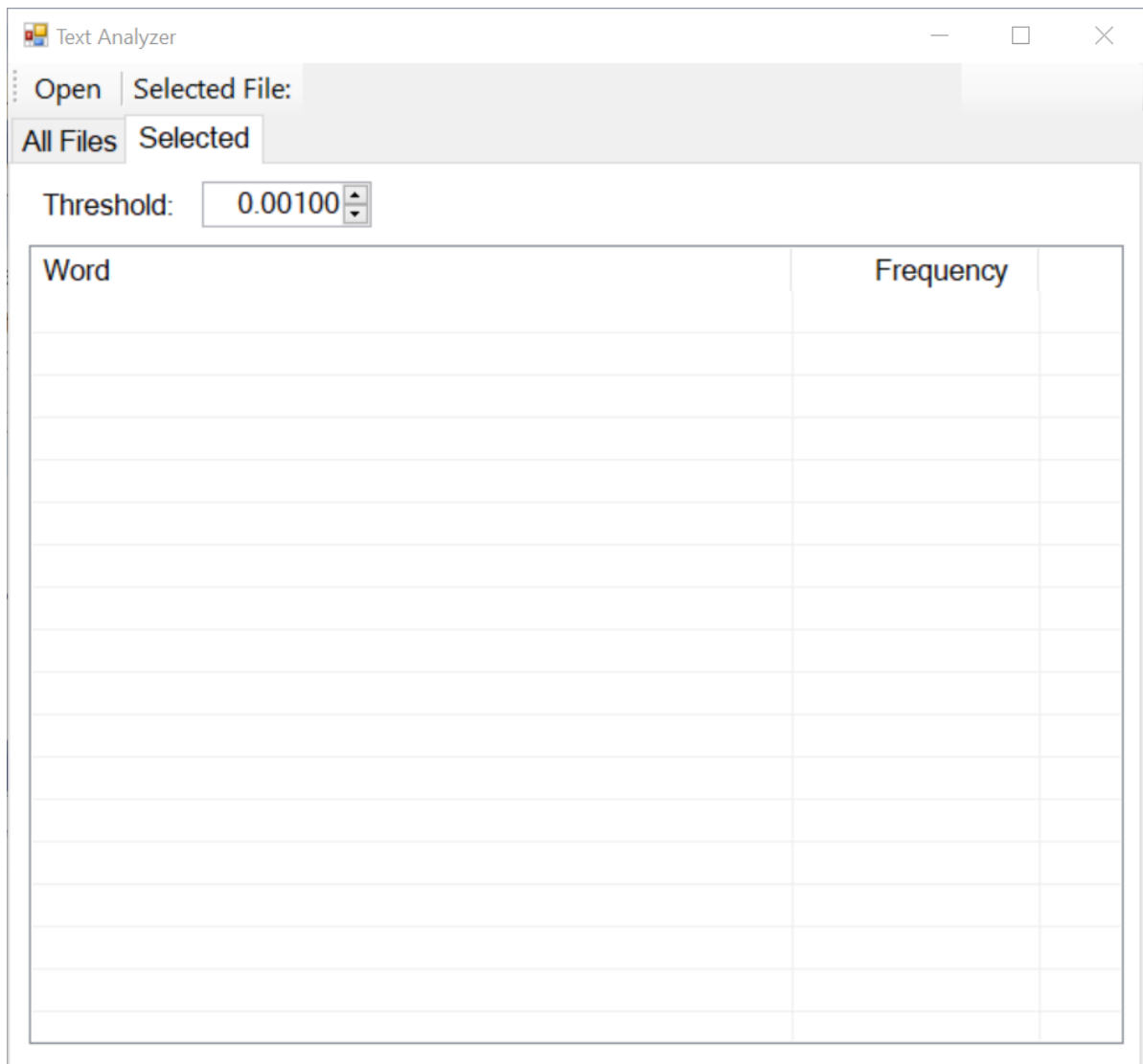
Below the label and the **NumericUpDown** is a [ListView](#), which is the control used by the Windows File Explorer to display the files in a folder. You will use it to display other information about the files in a folder. This **ListView** should fill most of the remainder of the **TabPage**. In order to cause the **ListView** to grow or shrink as the form is resized, showing scrollbars as needed, click its **Anchor** property, and on the drop-down to its right, click the small rectangles at the right and the bottom, so that all four small rectangles are dark gray. This will anchor the control to all four sides of the **TabPage**. Also, set the following properties as indicated:

- **GridLines: True**. This will cause lines to be drawn between the rows and columns.
- **MultiSelect: False**. This will limit the selection to a single item.
- **View: Details**. This will cause the items to be shown as lines of text in multiple columns.

To set up the columns of the **ListView**, click the "Edit Columns..." link at the bottom of its Properties window. Use the "Add" button in the resulting dialog to create a total of three members, which will correspond to the three columns. For each member, use the Properties window at the right of the dialog to give the member an appropriate name (following the [naming conventions](#) for this class) and to set the column width. (Note that you can also adjust the column widths by dragging the lines separating columns in the Design window; however, this doesn't seem to work very well - only changes within the ColumnHeader Collection Editor dialog are reliably reflected by the program.) For the second and third columns, set the **TextAlign** property to **Right** so that the columns are right-aligned.

Don't worry about setting the text for the column headings here - you will do that with program code (see "Coding Requirements" below). You will probably need to adjust the column widths after you have completed the code - each needs to be wide enough to display the entire heading when the column is sorted. Note that the user will also be able to change column widths by dragging the lines separating the columns, but the initial widths should be large enough to make this unnecessary.

To get to the second **TabPage**, you will need to use the **Tab** key to cycle through the controls. Within the running program, this page should initially look something like this:



Thus, the layout is similar to the first tab. As with the **ListView** in the first tab, you don't need to enter the column headings - this will be done by your code. Set the **TextAlign** property of the second column to **Right**. For the **NumericUpDown**:

- Set its range to be 0 to 1.
- Set its **DecimalPlaces** property to 5.
- Set its initial value to 0.00100.
- Set its increment to 0.001.


Apart from the visible controls, the GUI also needs a [FolderBrowserDialog](#).

Behavior of the GUI


This section details the behavior that you will need to implement, as described under "Coding Requirements" below (see the [demo video](#) for examples). Before discussing this behavior, we will note that each of the column headings in the two **ListViews** should contain a 3-character suffix. Each of these suffixes is initially 3 blank characters; hence, the headings on the right-aligned columns will not appear to be fully aligned to the right. Otherwise, the initial suffixes won't be visible.

Clicking the "Open" button should cause the **FolderBrowserDialog** to be displayed. If the user cancels the dialog, nothing in the main GUI should change. If the user selects a folder, the program should attempt to read each file in the selected folder, ignoring any sub-folders, and find the total number of distinct words in each file, along with the frequency of each distinct word occurring in each file (see under "User Requirements" above for the definitions of a word and the





frequency of a word in a file). If any exception occurs during the reading of these files this exception should be displayed in a **MessageBox**, and nothing in the GUI should change - all information maintained by the program should remain as it was prior to the selection of the folder.

If a folder is read successfully, any text in the text box within the **ToolStrip** should be removed, and the "All Files" tab should be displayed. Its **NumericUpDown** should contain whatever value was last entered into it, or its initial value if no value has been entered. The **ListView** on this tab should contain the names of all files in the folder (*not* the full path names) listed alphabetically in the "File Name" column. The suffix on the "File Name" heading should be set to two blank characters followed by  to indicate that the rows of the **ListView** are sorted by this column in ascending order. Each row of the "Vocabulary" column should contain, right-aligned, the number of distinct words in the file named on that row. The "Difference" column should be empty. The suffixes on the "Vocabulary" and "Difference" column headers should be three blanks. The "Selected" tab, if selected by the user, should contain in its **NumericUpDown** whatever value was last entered (or its initial value if no value has been entered), and should contain in its **ListView** its initial (empty) contents, as shown above.

If a file name in the first column of the "All Files" **ListView** is clicked, any text in the text box in the **ToolStrip** should be replaced by the selected file name. In each row of this **ListView**, the difference measure (see under "User Requirements" above) between the selected file and the file named on that row should be shown right-aligned in the "Difference" column. Each difference measure should be computed using the value in this tab's **NumericUpDown** as n . These values should be shown using the format, "9.99999" (i.e., showing five digits to the right of the decimal point). If the two files have no words in common, " ∞ " should be shown. Otherwise, this tab should remain unchanged.

Whenever a file name is clicked, the "Selected" tab should also change, although it won't be shown until the user selects it. Specifically, each word having a frequency of at least the threshold value (in this tab's **NumericUpDown**) within the selected file should be listed alphabetically in the "Word" column of this tab's **ListView**. The suffix for the "Word" heading should be two blank characters followed by  to indicate that this column is sorted in ascending order. In each row, the frequency of the word shown in that row within the selected file should be shown right-aligned in the "Frequency" column. These values should be formatted in the same way as the difference measures in the other **ListView**. The suffix for the "Frequency" heading should be three blank characters.

Clicking any column heading in either **ListView** should have the following effect:

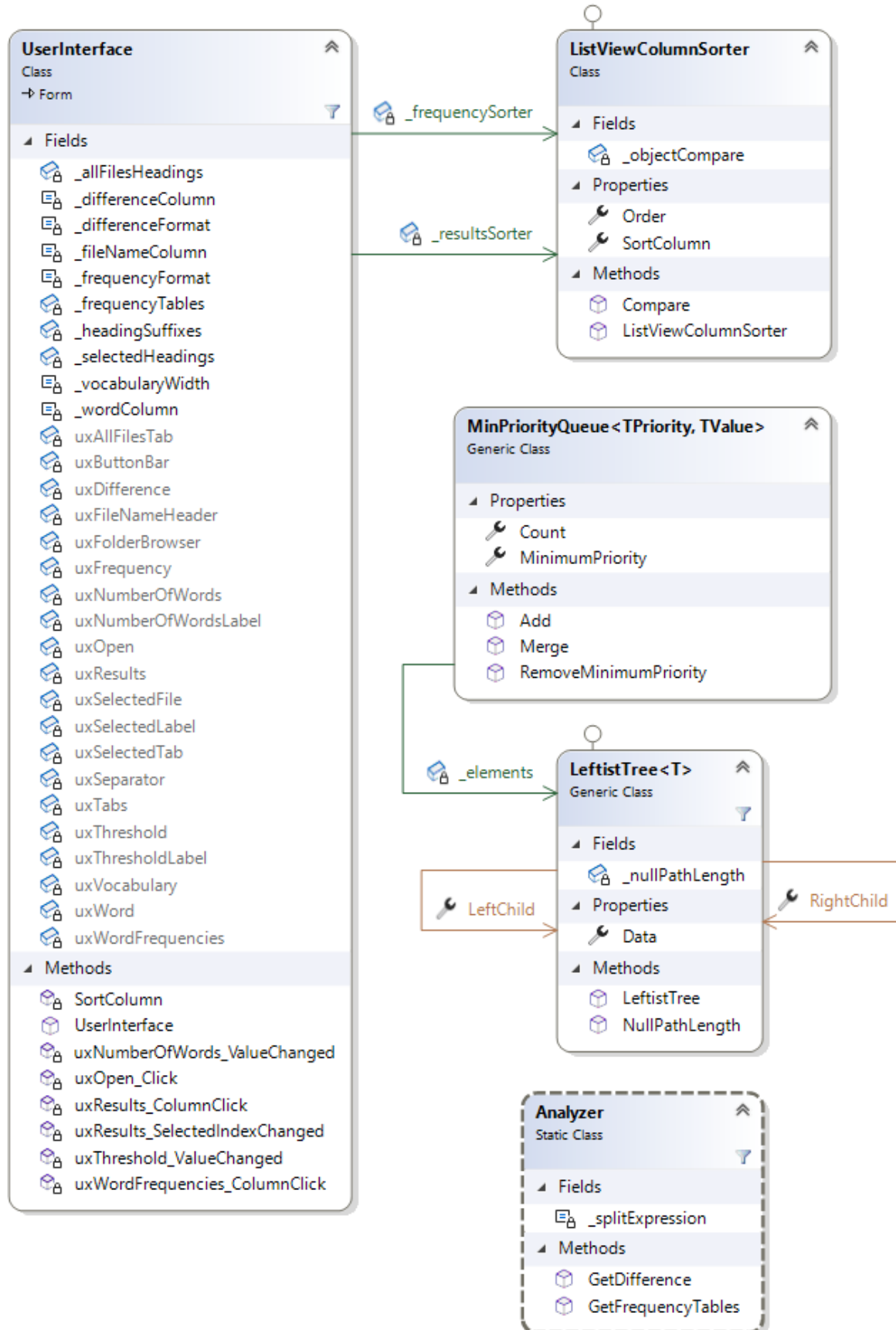
- If the suffix for this column heading end in  (indicating that this column is sorted in ascending order), then the rows of this **ListView** should be sorted by this column in descending order. The  in the column heading suffix should be replaced by  to indicate that this column is in descending order.
- Otherwise, the rows of this **ListView** should be sorted by this column in ascending order. The suffix this column heading should be replaced by two blanks, followed by . The other column heading suffixes in this **ListView** should be set to three blanks.

Changing the value in the **NumericUpDown** on the "All Files" tab should only cause other changes if the text box in the **ToolStrip** contains a file name. In this case, the difference measures are recomputed using the new **NumericUpDown** value as n . The "Difference" heading suffix should be set to three blanks to indicate that this column may no longer be sorted.

Changing the value in the **NumericUpDown** on the "Selected" tab should cause the contents of the **ListView** on this tab to be replaced by the words whose frequencies are at least the new value, along with these frequencies. The rows should be sorted as indicated by the column header suffixes (i.e., the sort column and order should be the same as it was before this change).

Software Architecture

The software architecture is shown in the following class diagram:



The **UserInterface** class implements the GUI. You will need to make additions to this class. You will also need to add the **static** class **Analyzer**, which will contain methods for doing the text analysis. You will not need to change the three provided classes, **ListViewColumnSorter** (which defines how comparisons are done when sorting a **ListView**) **MinPriorityQueue<TPriority, TValue>** (which you can use as you did for Lab Assignment 26), or **LeftistTree<T>** (which is only used by **MinPriorityQueue<TPriority, TValue>**).

We will describe the **ListViewColumnSorter** class in the next section. We will describe the code you will need to write for the **UserInterface** and **Analyzer** classes in the "Coding Requirements" section.

Data Structures

In this section, we will describe the main data structures you will need to use.

Frequency Tables

For each file in the selected folder, you will need to maintain a frequency table giving the frequency of each word appearing in the file. To implement each of these frequency tables, you will use a **Dictionary<string, float>**, where each word from the file is a key in the dictionary, and its associated value is its frequency in the file. You will need to keep all these frequency tables in a **Dictionary<string, Dictionary<string, float>>**. Each key in this dictionary is the name of a file (not the full path name), and its associated value is the frequency table for this file.

The ListView

Although the **ListView** is a GUI control, it is complex enough to be considered a data structure as well. Here, we discuss how you will need to use it.

The column headers of a **ListView** are accessed separately from the rows below them. Specifically, they are accessed via the [Columns](#) property. This property gets a collection that can be indexed like a singly-dimensioned array. To access the text displayed in an individual column header, use its [Text](#) property, which gets or sets a **string** giving the displayed text. Thus, for example, to set the text of column 0 (the first column) of a **ListView** `lv` to the contents of the **string** `s`:

```
lv.Columns[0].Text = s;
```

To access the rows below the column headers, use the **ListView**'s [Items](#) property. This property also gets a collection that can be indexed like a singly-dimensioned array. You can also add rows by using the **Items** property's [Add](#) method, which takes a [ListViewItem](#) as its only parameter. You can construct a **ListViewItem** whose first column contains a given **string** by passing that **string** to its [constructor](#). All rows can be removed using the **Items** property's [Clear](#) method.

To access the contents of a row, use the **ListViewItem**'s [SubItems](#) property. This property gets another collection that can be indexed like a singly-dimensioned array. When the **ListViewItem** is constructed, this collection has only one element, the **string** that was passed to the constructor. (This **string** can also be accessed via the **ListViewItem**'s [Text](#) property.) To add contents for the remaining columns, use the **SubItems** property's [Add](#) method, which takes a **string** as its only parameter. You can then get or set the text of a subitem at a particular index within the collection using its [Text](#) property.

To sort the rows of a **ListView**, use its [Sort](#) method. The way the rows are compared when they are sorted is defined by the **ListView**'s [ListViewItemSorter](#) property, which gets or sets the [IComparer](#) used to do the comparisons. The **IComparer** interface is similar to the **IComparable<T>** interface, but instead of having a **CompareTo** method, it has a **Compare** method that compares two **objects**. The **ListViewColumnSorter** class has been provided in your start code to serve as this **IComparer**. It has two properties that can be set to control how the comparisons are made:

- **SortColumn**: gets or sets an **int** giving the column on which the sorting is to be done.
- **Order**: gets or sets a [SortOrder](#) indicating in what order the sorting is to be done. **SortOrder** is an enumeration containing the following values:
 - **SortOrder.None**: no sorting is to be done. The underlying value 0.
 - **SortOrder.Ascending**: sort in ascending order. The underlying value is 1.
 - **SortOrder.Descending**: sort in descending order. The underlying value is 2.

Coding Requirements

In this section, we give the specific coding requirements for the **UserInterface** and **Analyzer** classes. You are not required to use the same names as shown in the class diagram as long as you follow the [naming conventions](#) for this course. For each class, you will be required to break the code into more **private** methods than are described below or shown in the class diagram above. Look for places where duplicate code can be avoided by putting it into a separate method, or where complex methods can be broken into simpler ones. You are free to break the code into more **private** methods than are required.

The Analyzer Class

Note that because this class is **static**, all members must be either **static** or **const**. You must include a **private const string** storing the value "[^a-z]+". This string is a [regular expression](#) describing all nonempty strings containing no lower-case English letters. You can use this string to break a string of text into words, as described below. You must also include two **public** methods and at least three **private** methods. The **public** methods are described in what follows.

A method to get the frequency tables

This method should take as its only parameter a **string** giving the full path name of the folder containing the files to process. It should return a **Dictionary<string, Dictionary<string, float>>**, as described under "Frequency Tables" above. From the given path name, you will need to get a [DirectoryInfo](#) describing the folder (refer to Lab Assignment 16 to review the **DirectoryInfo** and **FileInfo** classes).

For each file in the folder:

- Open the file using the **FileInfo**'s [FullName](#) property.
- Read the file a line at a time, counting the number of words in the file and counting the number of occurrences on each distinct word using a **Dictionary<string, int>** (this dictionary will function like the **long[]** in Lab Assignment 26 - each word in the file is a key, and its associated value is its number of occurrences in the file).

After converting a line of text to lower-case, you can break it into a **string[]** of individual words using the **static** method [Regex.Split](#) (in the **System.Text.RegularExpressions** namespace) using the string constant described above as its parameter. Note, however, that if the text line begins or ends with a non-letter, one or two of these array elements may be empty strings - you will need to ignore these elements.

- Using the **Dictionary<string, int>** created above, build a frequency table (see "Frequency Tables" above) in a **Dictionary<string, float>**, and add it to the **Dictionary<string, Dictionary<string, float>** you are building. The key for each frequency table in this last dictionary should be the name of the file it describes, obtained using **FileInfo's** [Name](#) property.

Don't do any exception handling within this method.

A method to compute the difference measure

This method should take as its parameters two **Dictionary<string, float>**s, each giving a frequency table (see "Frequency Tables" above) for a file, and an **int** giving the maximum number of words to use in computing the difference (i.e., the value n from the section, "User Requirements", above). It should return a **float** giving the difference measure.

Use a **MinPriorityQueue** to accumulate the words occurring most frequently. For each word in one of the given dictionaries, look it up in the other dictionary. If it is there, find the minimum of its frequencies in the two dictionaries. Use this minimum as its priority in the **MinPriorityQueue**. If adding this word causes the number of words in the queue to exceed the maximum number of words, remove the one with minimum priority.

At this point, your **MinPriorityQueue** should contain the words you need to use in computing the difference measure. Be sure to return ∞ if the queue is empty. Otherwise, compute the difference measure using all the words in the queue.

The UserInterface Class

To this class, you will need to add the following **private** fields:

- **const ints** defining the column indices for the file names and the differences in "All Files" **ListView** and the column index for the words in the "Selected" **ListView**. You won't need to reference any of the other columns specifically. In your code, use these fields rather than hard-coding the numbers, as this makes the code more maintainable.
- **const strings** defining [format strings](#) to format the difference measures and the frequencies, respectively, shown in the **ListViews**. Both of these constants should be set to "N5". One of these constants should be passed as the parameter to the [ToString](#) method when you convert one of these **floats** to a string. This will cause the value to be formatted as described under "Behavior of the GUI" above.
- A **const int** giving the number of characters to use when formatting a value in the "Vocabulary" column. Set this value to 10. We need to format these values because when we sort based on the values in any column, we are sorting strings. This causes problems when sorting by the "Vocabulary" column when values contain a different number of digits. For example "10" will be found to be less than "5" based on the comparison of their first characters ('1' is less than '5'). In order to fix this problem, we need to pad each string on the left by enough blanks so that all strings have the same length (a blank is less than any digit character). This can be done using [string interpolation](#). For example, suppose the name of this constant is `_vocabularywidth`, and let `n` be an **int** variable. Then the following line of code will cause `s` to contain `n` converted to a **string**, padded on the left with blanks so that its length is 10:

```
string s = $"{n, _vocabularywidth}";
```

- **string[]** s containing the base column headers (without suffixes) for each of the **ListViews**. List the headings in the order they will appear in each **ListView**.

- A **string[]** giving the column header suffixes:

```
private string[] _headingSuffixes = { " ", " ", " ^", " ~" };
```

Note that the index of each suffix is the same as the underlying value of the **SortOrder** member corresponding to that suffix. Listing these suffixes in this order will allow you to use **SortOrder** values to index the appropriate suffix; e.g., if the **SortOrder** variable `order` indicates the order in which a column is sorted, you can access the appropriate suffix for that column's heading as follows:

```
string suffix = _headingSuffixes[(int)order];
```

- A **Dictionary<string, Dictionary<string, float>>** to refer to the frequency tables. There is no need to initialize it here.
- A **ListViewColumnSorter** for each of the **ListView**s. You can initialize these to new instances.

You will also need six event handlers and at least four other **private** methods, and you will need to modify the constructor. In what follows, we describe the modifications to the constructor as well as all the event handlers and one of the four other **private** methods.

A method to sort a ListView by a given column in a given order

This method should take the following parameters:

- The **ListView** to sort.
- A **string[]** giving the base column headings for this **ListView**.
- An **int** giving the index of the column to sort.
- A **SortOrder** giving the order in which to sort the column.

It should return nothing. It is responsible for sorting the **ListView** as specified and setting all of its column headings appropriately.

You will first need to get the **ListView**'s **ListViewItemSorter** property and cast it to **ListViewColumnSorter**. Then set its **SortColumn** and **Order** properties appropriately, and if the given **SortOrder** isn't **SortOrder.None**, sort the **ListView**. Then set each of its column headings appropriately. (Don't try to use whatever heading might already be there - just construct the appropriate heading and replace whatever is there.)

You should use this method whenever you need to update the column headings for either **ListView**. This will help to keep the properties of each **ListView**'s **ListViewItemSorter** consistent with the heading suffixes. If you don't need to sort, use **SortOrder.None** as the last parameter.

The constructor

After the call to **InitializeComponent**, you will need to set each **ListView**'s **ListViewItemSorter** property to the appropriate **ListViewColumnSorter** field and sort each **ListView** using the above method in order to set the column headings (use **SortOrder.None**).

The event handler for the "Open" button

Within this event handler you will need to implement the functionality of the "Open" button, as described under "Behavior of the GUI" above. Display the **FolderBrowserDialog** as you would a file dialog. If the user selects a folder, you can obtain it from the **FolderBrowserDialog**'s [SelectedPath](#) property, which gets or sets a **string** giving the path selected by the user. See

under "The **ListView**" above for how to interact with the **ListView**s. Add items to the "All Files" **ListView** by iterating through the frequency tables to obtain the information you need. Don't worry about the order that you add the items - you can sort them after they are added. As you add items, add enough subitems for all of the columns, even though one will be empty at this point (just use the empty string as its text). To select the tab to be displayed, use the **TabControl**'s [SelectTab](#) method, which takes as its only parameter the **TabPage** to display. Be sure to handle any exceptions that may be thrown during the reading of the files.

An event handler for the "All Files" ListView

For this **ListView** you will need to create a default event handler (i.e., by double-clicking the **ListView** in the Design window). This event handler will handle **SelectedIndexChanged** events, which are signaled when the user clicks an item in the first column (i.e., a file name). When this happens, two **SelectedIndexChanged** events may be signaled - one to remove any previously-selected index, and one to add the newly selected index. You will therefore need to make sure that the **ListView**'s [SelectedItems](#) property, which gets an **IList** containing the selected items, is not empty. If this is the case, it will contain a single **ListViewItem**, which will be the item selected by the user.

Implement the behavior described under "Behavior of the GUI" above. Depending on the value in the "Threshold" **NumericUpDown**, the number of items added to the "Selected" **ListView** may be rather large. To improve the efficiency of a potentially large number of updates to this GUI control, you should call its [BeginUpdate](#) method before making any changes to this GUI, and call its [EndUpdate](#) method after all changes have been made. This prevents the control from being drawn until after all changes have been made. Each of these methods takes an empty parameter list.

An event handler for the "Number of Words Used" NumericUpDown

This event handler will be called whenever the value in this **NumericUpDown** is changed. Implement its behavior as described under "Behavior of the GUI" above.

A ColumnClick event handler for the "All Files" ListView

You will need to create this event handler through the Events tab of the Properties window within the Design window. It will be called whenever one of the column headings in the **ListView** is clicked. Its [ColumnClickEventArgs](#) parameter contains information about the event. In particular, its [Column](#) property gets the index of the column that was clicked.

You will need to implement the behavior specified under "Behavior of the GUI" above. See ["Use Visual C# to sort a ListView control by using a column"](#), Step 6, for sample code for an event handler having similar behavior. Note that this code only sorts the **ListView** - it doesn't change the column headings, as your code will need to do. You may copy this code, but you will need to modify it to fit your context.

An event handler for the "Threshold" NumericUpDown

This event handler will be called whenever the value in this **NumericUpDown** is changed. Implement its behavior as described under "Behavior of the GUI" above.

A ColumnClick event handler for the "Selected" ListView

This event handler will be called whenever one of the column headings in this **ListView** is clicked. Implement its behavior as described under "Behavior of the GUI" above.

Testing and Performance

Sample test data can be found in [this ZIP archive](#) (which you will need to decompress). Don't place this folder in your repository. The **Data** folder contains 12 files, each of which is the complete text of a book by one of three authors:

- Charles Dickens:
 - GreatExpectations.txt: *Great Expectations*
 - DavidCopperfield.txt: *David Copperfield*
 - ATaleOfTwoCities.txt: *A Tale of Two Cities: A Story of the French Revolution*
 - AChristmasCarol.txt: *A Christmas Carol in Prose: Being a Ghost Story of Christmas*
- William Shakespeare:
 - AMidsummerNightsDream.txt: *A Midsummer Night's Dream*
 - Macbeth.txt: *Macbeth*
 - Hamlet.txt: *Hamlet*
 - RomeoAndJuliet.txt: *Romeo and Juliet*
- Lewis Carroll:
 - AlicesAdventuresInWonderland.txt: *Alice's Adventures in Wonderland*
 - TheHuntingOfTheSnark.txt: *The Hunting of the Snark: An Agony in Eight Fits*
 - SymbolicLogic.txt: *Symbolic Logic*
 - ThroughTheLookingGlass.txt: *Through the Looking Glass: And What Alice Found There*

Its **Alt** subfolder contains the following files:

- banana.txt: The single word, "banana".
- bonnie.txt: The chorus to "My Bonnie Lies Over the Ocean".
- Hamlet.txt: A copy of Hamlet.txt from the **Data** folder.
- Hamlet-edited.txt: The above file with a single word, "by", removed.
- snark-paragraph.txt: The paragraph from *The Hunting of the Snark* quoted under "User Requirements" above.
- TheHuntingOfTheSnark.txt: A copy of TheHuntingOfTheSnark.txt from the **Data** folder.

The [demo video](#) shows several tests that you can perform on the above data and indicates the performance you should expect. Be sure to check all the values shown in the video, even if they aren't explicitly mentioned. To test exception handling, try opening a folder that you are not authorized to access (see Lab Assignment 16, Step 8, if you need to create such a folder)

Submitting Your Assignment

Be sure to **commit** all your changes, then **push** your commits to your GitHub repository. Then submit the *entire URL* of the commit that you want graded.

Important: If the URL you submit does not contain the 40-hex-digit fingerprint of the commit you want graded, **you will receive a 0**, as this fingerprint is the only way we can verify that you completed your code prior to submitting your assignment. We will only grade the source code that is included in the commit that you submit. Therefore, be sure that the commit on GitHub contains all seven .cs files within the **Ksu.Cis300.TextAnalysis** folder, and that it is the version you want graded. This is especially important if you had any trouble committing or pushing your code.

Upper Bound on the Difference Measure

In this section, for the benefit of those who are interested, we prove that the difference measure, if finite, is always less than $\sqrt{2}$.

Suppose x_1, \dots, x_k and y_1, \dots, y_k are the nonzero frequencies of k words in two files, where k is at least 1. Recall that the difference measure is defined as:

$$d = \sqrt{(x_1 - y_1)^2 + \dots + (x_k - y_k)^2}.$$

Because each x_i and y_i is a nonzero frequency, we have $0 < x_i \leq 1$ and $0 < y_i \leq 1$ for $1 \leq i \leq k$.

Then $|x_i - y_i| < 1$. Multiplying both sides of this inequality by $|x_i - y_i|$, we have

$$\begin{aligned} |x_i - y_i|^2 &< |x_i - y_i| \\ (x_i - y_i)^2 &< |x_i - y_i| \end{aligned}$$

for each i .

Because each x_i is the frequency of a different word in the file, the sum of all the x_i s (and likewise of all the y_i s) is no more than 1. Thus, the sum of all the terms in which $x_i > y_i$ is less than 1, and the sum of all the terms in which $x_i < y_i$ is also less than 1. Because all other terms are 0, $d < \sqrt{2}$.