Paper, code, and example experiment by: Kathleen Napier
Collaborative Project 2

Text Analysis and C++ Programming Project for CS 1

By: Kathleen Napier, Gordon Stein, and Dr. Lior Shamir

College of Arts and Sciences, Lawrence Technological University

Contents

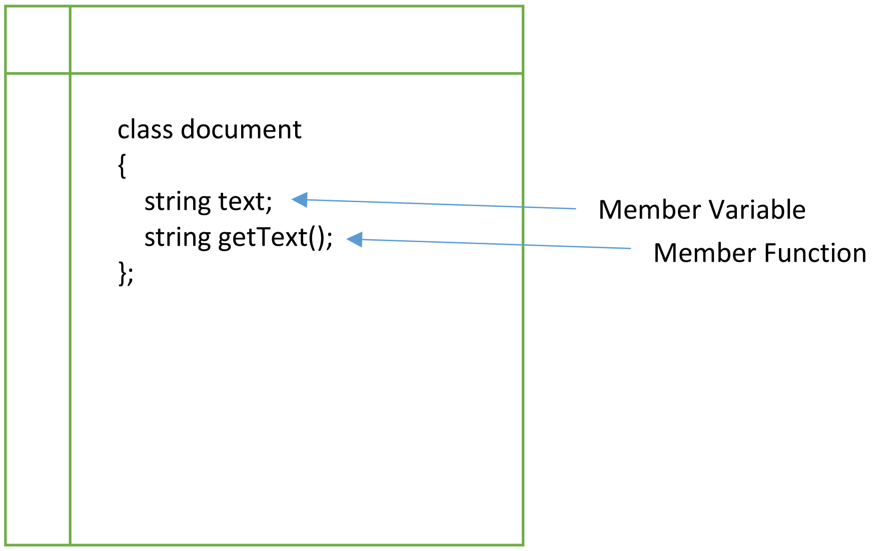
Introduction

In this manual, you are going to have the opportunity to complete a small research experiment on a topic that interests you, while learning the basics of C++. The text analysis project specified in these pages allows for the personalization of data choice. Due to this, you can analyze text by your favorite author, musician, blogger, etc. in order to learn a little more about the text features of their words, as well as the basics of text analysis. Before jumping into the actual analysis, let's beginning with some simple programming.

Classes

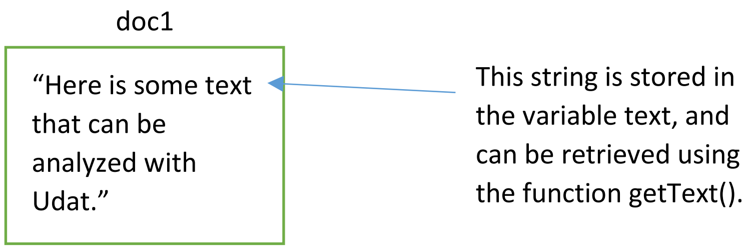
A variety of programming languages (such as C++, Java, Python, etc.) are object-oriented programming languages. They use user-defined objects to represent real-world objects. Each **object** is an instance of a class, and a **class** is a data structure composed of both member variables and member functions. The class is essentially the blueprint for an object. When performing text analysis, each text document could be viewed as an object of a class called ***document*** (see Fig. 1).

Figure 1: document Class



```
class document
{
   string text;
   string getText();
};
```

Once the components of a class are defined, we can create a ***document*** object. Let's call it doc1 (Fig. 2).
Figure 2: doc1 Object



"Here is some text that can be analyzed with Udat."

This string is stored in the variable text, and can be retrieved using the function getText().

When defining classes, we may not want all variables or function to be accessible to the public. This distinction can be made through the use of a **public** and **private** section in the class definition (Fig. 3). The public and private keywords are referred to as **access-specifiers**.

Figure 3: Public and Private Class Members

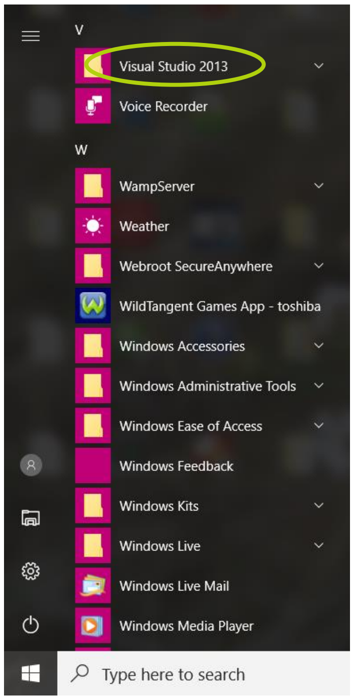
```
class document
{
   public:
      string getText();
   private:
      string text;
};
```

By making text private, it prohibits an outside user from modifying your original data.

<u>Building Program:</u>

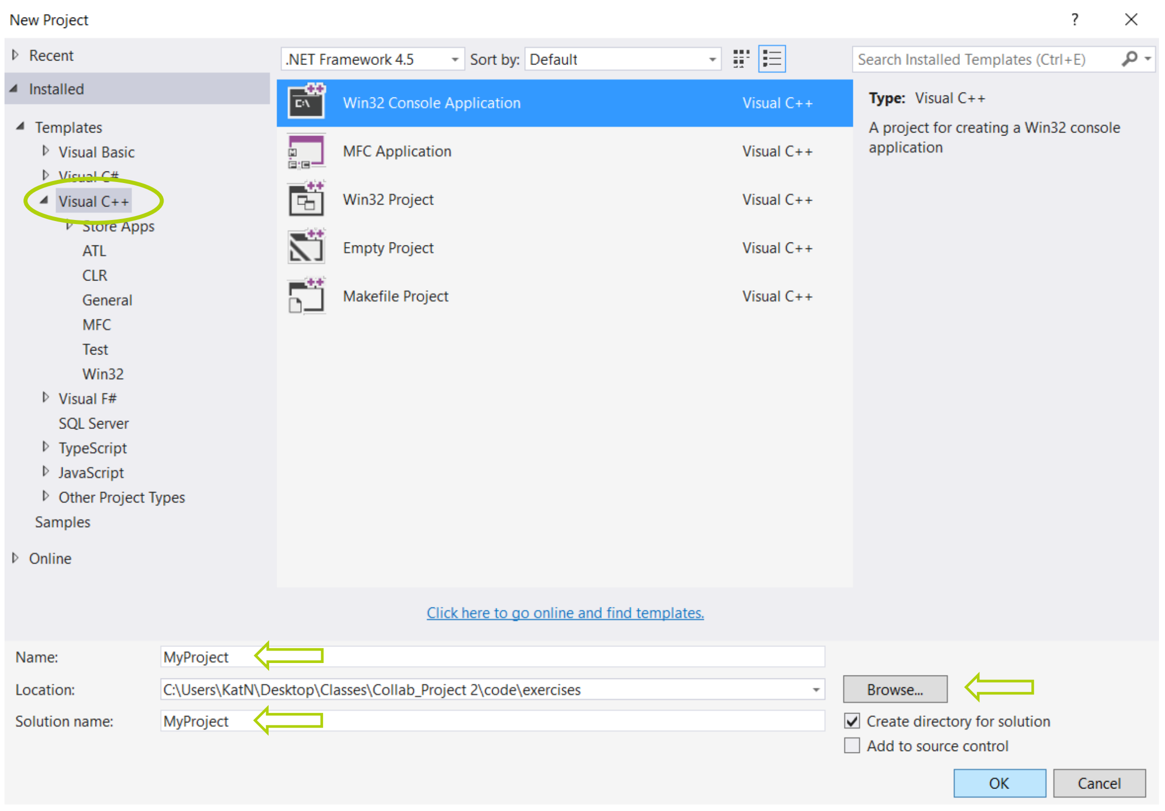
Try creating your own Document object.

1. Collect a small amount of text data. It could be a line from your favorite song, a sentence from your favorite book, or even a social media post.
2. Open up Visual Studio by either clicking the icon or finding Visual Studio in the start menu.



3. Go to the menu bar and click: File → New → Project

4. In the New Project window, select Visual C++ from the left side bar, and then select the Win32 Console Application. Rename the project as MyProject, or whatever name you would like. Click browse to determine where you would like to save your program, then select OK.


New Project

Recent
Installed
Templates
Visual Basic
Visual C#
Visual C++
Store Apps
ATL
CLR
General
MFC
Test
Win32
Visual F#
SQL Server
TypeScript
JavaScript
Other Project Types
Samples
Online

.NET Framework 4.5 | Sort by: Default | Search Installed Templates (Ctrl+E)

Win32 Console Application	Visual C++
MFC Application	Visual C++
Win32 Project	Visual C++
Empty Project	Visual C++
Makefile Project	Visual C++

Type: Visual C++
A project for creating a Win32 console application

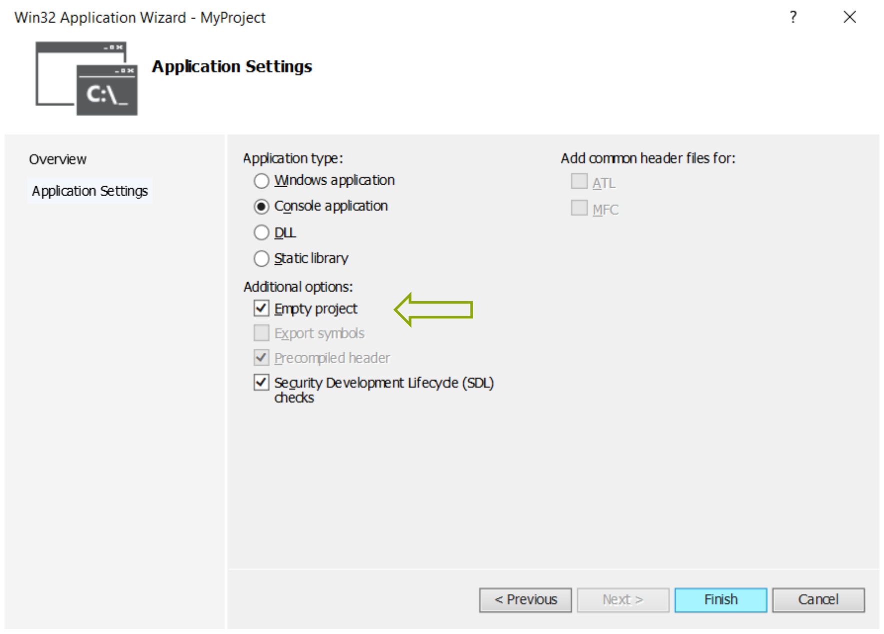
Click here to go online and find templates.

Name: MyProject
Location: C:\Users\KatN\Desktop\Classes\Collab_Project 2\code\exercises
Solution name: MyProject

Browse...
☑ Create directory for solution
☐ Add to source control

OK | Cancel

5. In the Win32 Application Wizard click Next. Select Empty project, and then click Finish.


Win32 Application Wizard - MyProject

Application Settings

Overview
Application Settings

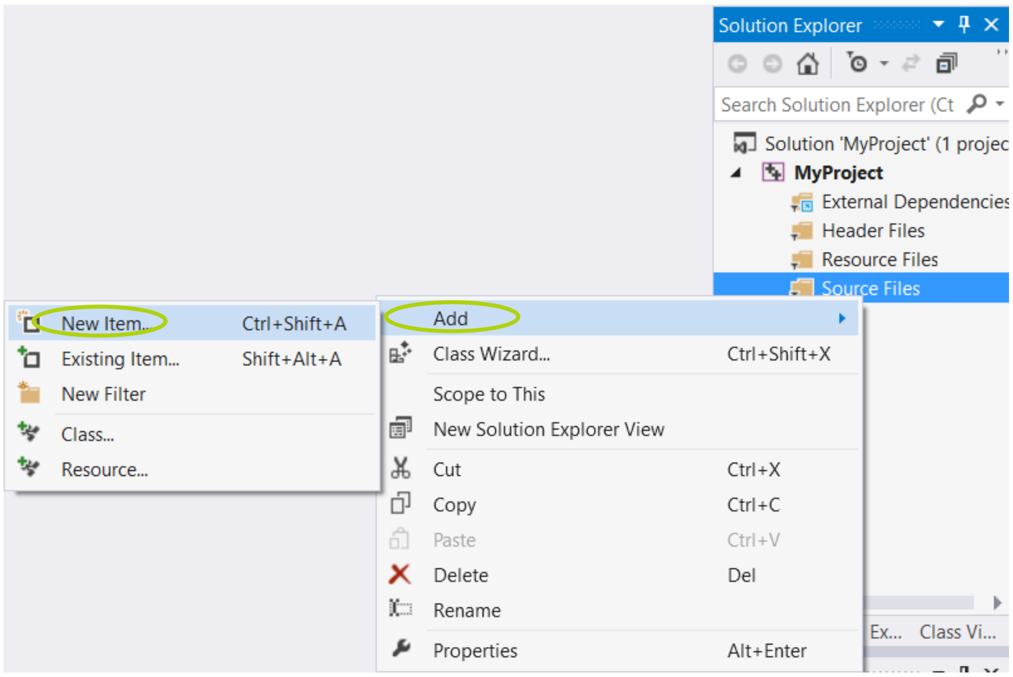
Application type:
○ Windows application
● Console application
○ DLL
○ Static library

Additional options:
☑ Empty project
☐ Export symbols
☑ Precompiled header
☑ Security Development Lifecycle (SDL) checks

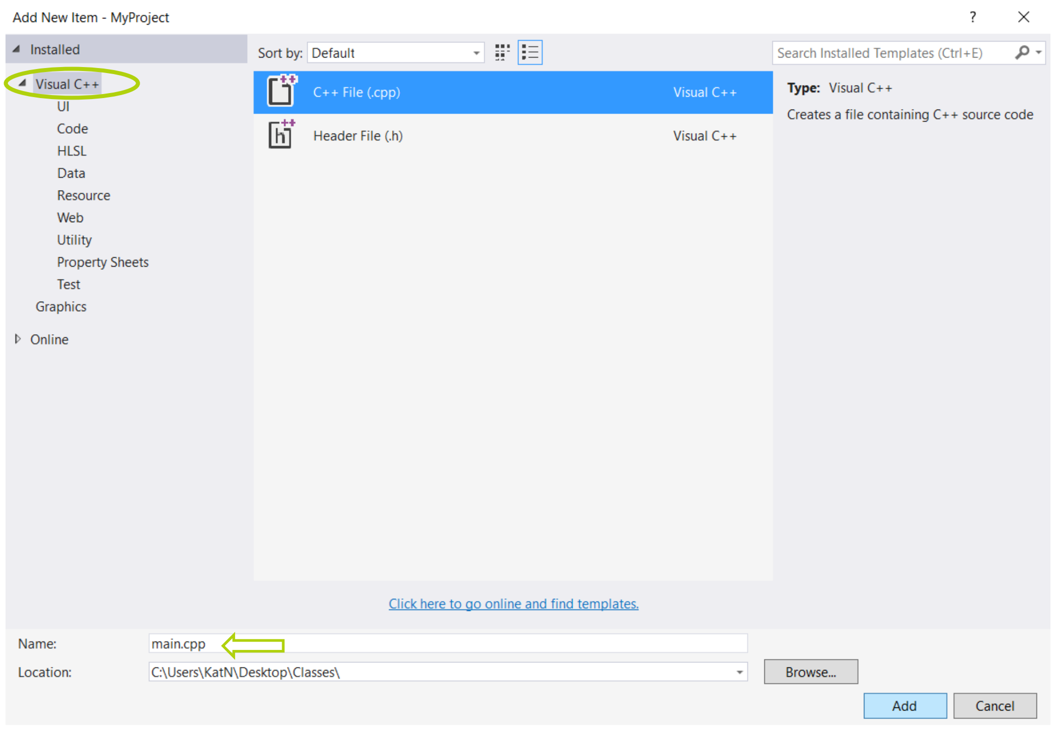
Add common header files for:
☐ ATL
☐ MFC

< Previous | Next > | Finish | Cancel

6. Locate the Solutions Explorer panel. It is usually in the upper right hand corner. If not open, select View from the menu bar and choose Solutions Explorer (or press Ctrl+Alt+L).
7. Create a new source file. Right click on Source Files in Solution Explorer, then select: Add →New Item…



In the Add New Item - Document window, select Visual C++ → C++ File (.cpp). Title the file main.cpp and click Add. Repeat to create document.cpp. Repeat once more, but select the Header File (.h) option to create a document.h file.


Add New Item - MyProject
Installed
Visual C++
UI
Code
HLSL
Data
Resource
Web
Utility
Property Sheets
Test
Graphics
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
C++ File (.cpp) Visual C++
Header File (.h) Visual C++
Type: Visual C++
Creates a file containing C++ source code
Click here to go online and find templates.
Browse…
Add
Cancel

Name: main.cpp
Location: C:\Users\KatN\Desktop\Classes\

8. Enter source code in document.h file to declare the member variables and functions of the class.

```
//document.h
#ifndef DOCUMENT_H_
#define DOCUMENT_H_
#include <string>

class document
{
   public:
      std::string getText() const;
      void setText( std::string s );
   private:
      std::string text;
};
#endif
```

Checks if DOCUMENT_H has been defined earlier or in an included file. If not, it defines DOCUMENT_H and continues with code.

9. Enter source code in document.cpp file to define the implementation of ***document's*** functions.

```
//document.cpp
#include “document.h”
using namespace std;

string document::getText() const
{
   return text;
}

void document::setText( string s )
{
   text = s;
}
```

Links class declaration to class definition.

10. Enter source code in main.cpp file.

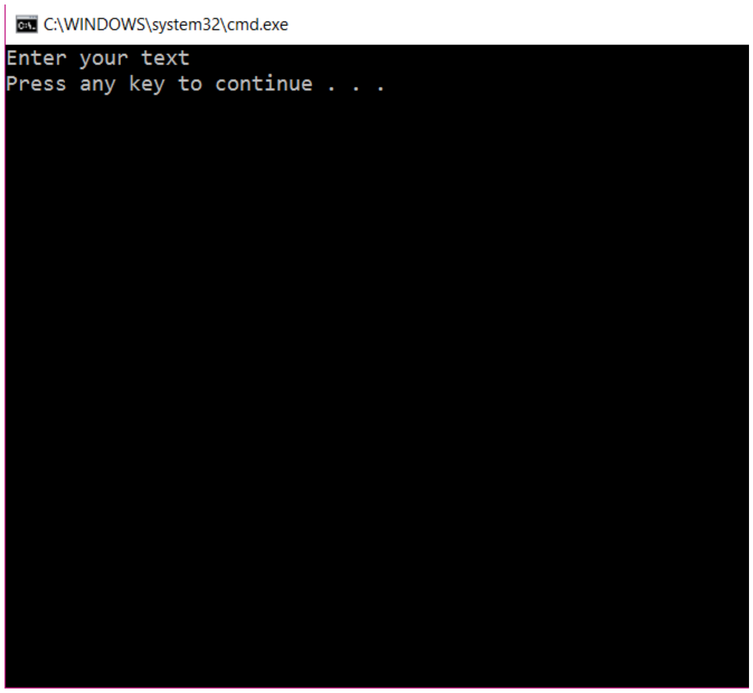
```
//main.cpp

#include <iostream>
#include “document.h”
using namespace std;

int main()
{
   document doc1;
   doc1.setText( “Enter your text” );
   cout << doc1.getText() << endl;
   return 0;
};
```

Enables program to print to screen.

Enables use of the ***document*** class.

To access a member variable or function of an object, use the dot operator (.) between the object’s name and the function’s or variable’s name.

11. Go to menu bar and click: File → Save All.
12. Go to Debug in the menu bar, and select Start Without Debugging (or press Ctrl+F5 on your keyboard). A pop-up window should open informing you that the Document project is out-of-date. Choose Yes, to build an updated version before running the program. Your program should print your line of text in the output window.



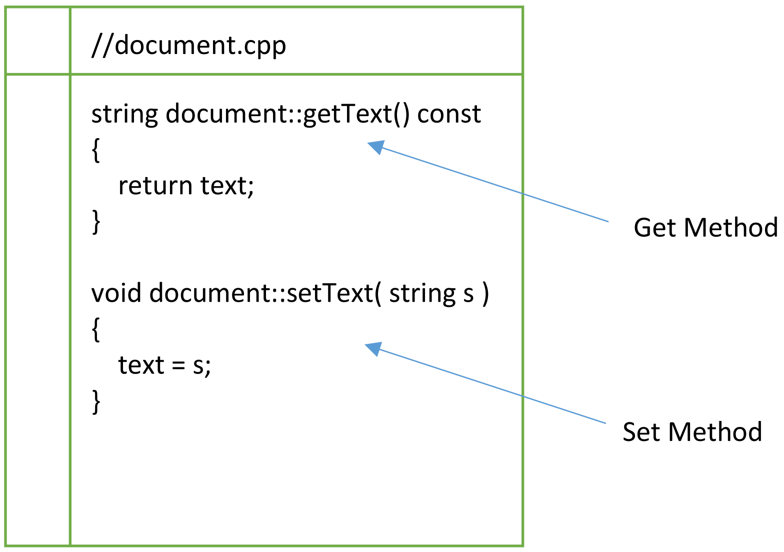
```
Enter your text
Press any key to continue . . .
```

More on Classes

You now have a basic definition for the class ***document***. Before moving on, it is important to highlight two concepts used in this class: get and set functions (see Fig. 1).

Get (accessor) functions enable public users to retrieve the value of private variables. A get function is often followed by the **const** keyword to signify that the members of the object will not be altered by this function. **Set (mutator) functions** enable public users to modify a private variable to a specified value. A set method may also include code to verify that the given value is appropriate for the private variable. For example, a set function setAge() might check that all input is non-negative before modifying the variable.

Figure 1: Get and Set Methods



```
//document.cpp

string document::getText() const
{
   return text;
}

void document::setText( string s )
{
   text = s;
}
```

To complete the general design of the ***document*** class we need to add constructor and destructor methods. A **constructor method** is called anytime a new object is created. It generates the new object instance for your program. In C++, each defined class automatically comes with a **default constructor**. So the ***document*** class you coded earlier had its own default constructor. This was why you were able to create the ***document*** instance doc1. When explicitly writing a default constructor, its syntax is similar to that of a function that: takes the name of the class, has no parameters, and has no return type (not even void). See Fig. 2 for this format.

Figure 2: Default Constructor

```
//document.h
```

```
class document
{
   public:
      document();
   private:
      std::string text;
};
```

Default Constructor

Constructors are used to perform actions necessary for the creation of an object, such as allocating memory for variables. However, a constructor will not set default values to built-in types (int, float, etc.). For specific values, they must be explicitly written by either the user or in the class definition. Thus, there may be times you will want to override a default constructor. See Fig. 3 for an overridden default constructor of the ***document*** class.

Figure 3: Override Default Constructor

```
//document.cpp
```

```
document::document()
{
   text = “default text”;
}
```

Additionally, constructors can be defined with parameters. In the ***document*** class we may want to create a document and set its text variable at the same time (Fig. 4).

Figure 4: Constructor with Single Parameter

```
//document.h

class document
{
   public:
      document( std::string s );
   private:
      std::string text;
};
```

```
//document.cpp

document::document( string s )
{
   text = s;
}
```

Now to create a new object, we need to call a constructor with an object declaration. Fig. 5 demonstrates an object declaration for both a default and parameterized constructor.

Figure 5: Object Declaration-Calling Constructors

```
document doc1;
document doc2( "test string" );
```

An alternative method of calling a constructor is to use the keyword **new** in an object's initialization, as demonstrated in Fig. 7. The key difference of using the new keyword is in the location where, in this case the object, is stored. This manual will not explore the intricacies of new; but note: as you use pointers in later sections, you will occasionally need to use the new keyword for object declaration.

Figure 7: Object Initialization-Calling Constructors

```
document * doc1 = new document();
document * doc2 = new document( “test string” );
```

This is an example of pointer use. We will explain pointers after reviewing arrays and strings.

Additionally, when you create an object, you need to delete that object in order to free up memory. A **destructor** function will inherently be called by the program when the destructor’s class goes out of scope or when the keyword **delete** is called on a pointer to the destructor’s class. The destructor does not take any parameters and returns nothing. It takes the form shown in Figure 8.

Figure 8: Destructor

```
//Formula:
//~className()

~document();
```

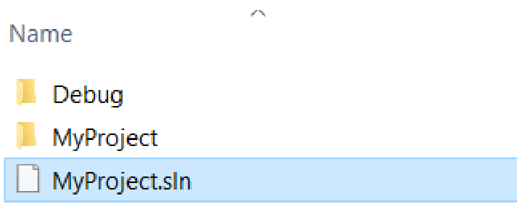
Like with constructors, classes also come with a default destructor. You should explicitly write the destructor if the class contains members that should be cleaned up with a destructor call or the delete keyword. The ***document*** class does not contain anything that should be deleted explicitly. We will cover more on the keyword delete in the Pointers section.

Debugging Tip: Once a constructor is explicitly defined in a class, the class’s original default constructor no longer exists. If calling the default constructor produces an error, double check your class definition. It may be the case that you are calling a default constructor that no longer exists.

Building Program:
Let’s expand on the ***document*** class:

1. Open up Visual Studio using the icon or the start menu .
2. Go to menu bar and click: File → Open → Project/Solution....

3. In the Open Project window locate your saved Document program. Enter its folder and select the MyProject.sln file (or the name you chose to save it as earlier). Click Open.



4. To add a default constructor and parameterized constructor that takes a single *string*; update the document.h and document.cpp files to match the code below. Code to be added is in bold.

```
//document.h

#ifndef DOCUMENT_H_
#define DOCUMENT_H_
#include <string>

class document
{
  public:
    document();
    document( std::string s );
    std::string getText() const;
    void setText( std::string s );
  private:
    std::string text;
};
#endif
```

```
//document.cpp

#include "document.h"
using namespace std;

document::document()
{
  text = "";
}

document::document( string s )
{
  text = s;
}

string document::getText() const
{
  return text;
}

void document::setText( string s )
{
  text = s;
}
```

5. Using the same line of text stored in doc1 or a new line of text (i.e. a quote from favorite movie, sentence from your favorite book, etc.), write code for the given instructions.
Coding Exercise: Based on the examples given for object declaration and initialization, update the main.cpp to include:
    a. A document object called doc2 declared with the parameterized constructor.
    b. A document object called doc3 declared using the new keyword and the default constructor.
See Appendix A-1 for an example solution.

6. Save your work and run program for debugging.

Arrays and Strings

As you begin your text analysis project, you will collect texts from a source of your choosing. Let's say you're a film buff, and you decide to collect quotes from the protagonists of various Spielberg films. This collection of quotes represents a set of items you will want to reference to throughout your project. So how can you store your text in a useable manner? There are various methods, such as in a text file or in a .csv file, but in this section we will discuss an in-code method: an **array**.

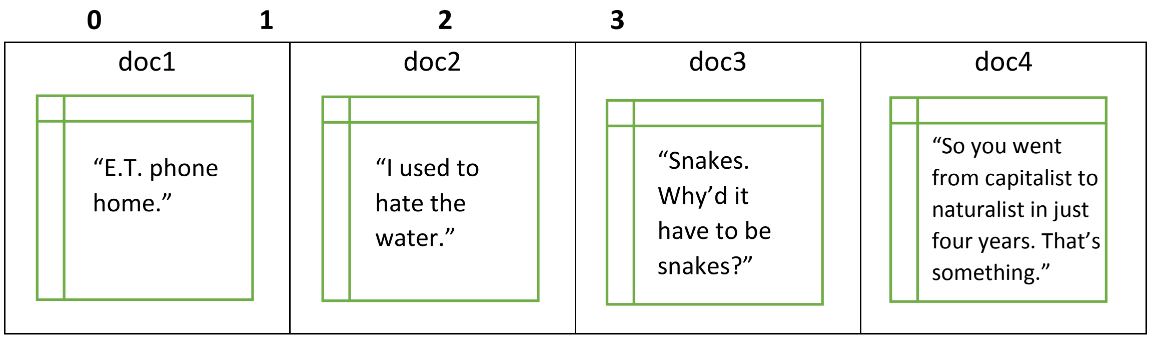
What is an array? It is a **data structure** that represents a set of elements. A data structure is a user– or language-defined data type that holds and arranges a group of data. Arrays can often be visualized as a set of storage containers, or boxes. Each box contains an element of the array. In terms of the example project from above, each box would contain a Spielberg quote. To declare this array, we need to: declare the type of the items in the array, the name of the new array, and the size of the array (Fig. 1).

Figure 1: Declaring an Array

```
//Array Declaration Formula:
// type name [size]

string quotes[4];
```

In the Fig. 1 we declared the quotes array to have a type of ***string***. In C++, a s*tring* is an object of the class type *string* that represents a unique arrangement of characters (**char**). An array does not need to contain only built-in data types. They can contain any pre-defined or programmer-defined objects. So you could make an array of the ***document*** objects that were defined earlier (Fig. 2).

Figure 2: Array of documents[1]


0
1
2
3
doc1
"E.T. phone home."
doc2
"I used to hate the water."
doc3
"Snakes. Why'd it have to be snakes?"
doc4
"So you went from capitalist to naturalist in just four years. That's something."

The class *string* essentially represents an array of char data types. For example, the char array in Fig. 3 represents the *string* "document".

Figure 3: Char Array Representing string Object

0	1	2	3	4	5	6	7
d	**o**	**c**	**u**	**m**	**e**	**n**	**t**

Later on in the section we will explore some basic functions of *string* class.

Continuing with our array of movie quotes, now that the array has been declared, see Fig. 4 for methods of initializing the array with a list of elements.

Figure 4: Array Initialization

```
string quotes[4] = {"q0", "q1", "q2", "q3"};

//OR

string quotes[] = {"q0", "q1", "q2", "q3"};
```

Stating the size of the array is not necessary if declaration occurs at the same time as initialization (as seen here). The compiler will assume the length of the array is equivalent to the number of items given in the initialization list.

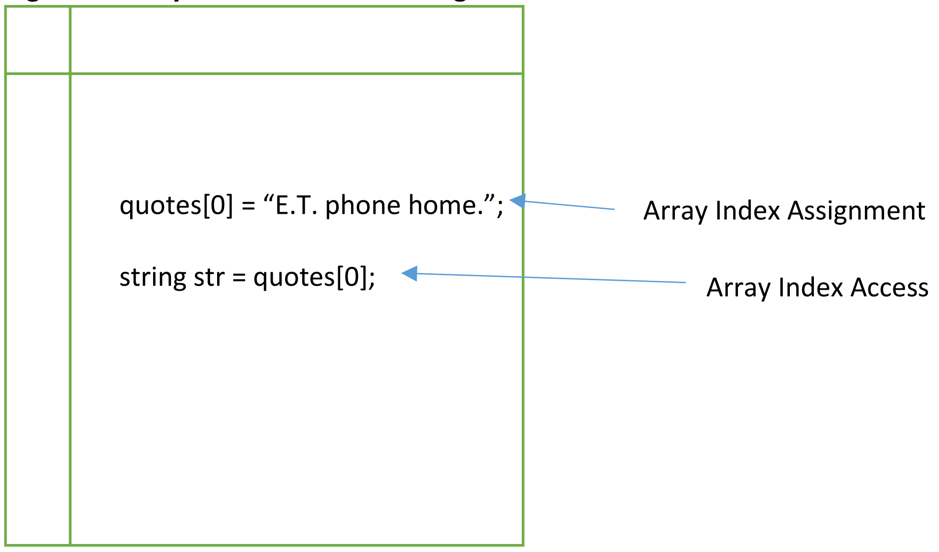
<u>Note:</u>

The initializer list of consecutive elements can be of a length less than the one defined in an array's declaration. Ex: int arr[4] = {"x "} where only arr[0] is initialized.

[1] Quotes in array are from Spielberg's films: E.T. the Extra-Terrestrial, Jaws, Raiders of the Lost Ark, and The Lost World: Jurassic Park.

Alternatively, the members of an array can be initialized individually following the code in Fig. 5. Figure 5 shows how to access and assign values of array indices.With these skills, you will now be able to reference an array of your stored texts later on in the main project.

Figure 5: Array Index Access and Assignment



```
quotes[0] = “E.T. phone home.”;

string str = quotes[0];
```

Remember:

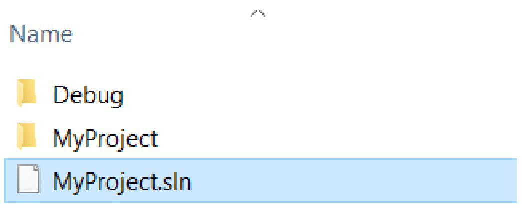
The indices of an array always start at 0.

For more information on what can be done with arrays check out the http://www.cplusplus.com website. They have a great tutorial on the page: http://www.cplusplus.com/doc/tutorial/arrays/

Building Program:

Follow the steps below to create an array of documents.

1. Open up Visual Studio using the icon or the start menu .
2. Go to menu bar and click: File → Open → Project/Solution....
3. In the Open Project window, locate your project’s saved program. Enter its folder and select MyProject.sln (or the solution file of the name you chose for your project). Click Open.



4. Modify main.cpp by declaring a ***document*** array of size 5, and then initializing the first two indices with the samples of text that you used in the previous sections. Print out second index of the array . New code is in bold.

```
//main.cpp

#include <iostream>
#include “document.h”
using namespace std;

int main()
{
    document doc1;
    doc1.setText(“ Enter your text ”);
    cout << doc1.getText() << endl;

    document doc2( “Enter new text” );
    document * doc3 = new document();

   document collection[5];
   collection[0] = doc1;
   collection[1] = doc2;
   cout << collection[1].getText() << endl;

    return 0;
};
```

   Is there another way to initialize the first two indices? Think about it, then check the alternative example solution in Appendix B-1.
5. Save your work and run program for debugging.

String Functions

In your ***document*** class, you stored text within a *string* object. However, *strings* are more than just text holders. The *string* class definition has a variety of member variables and functions that can be referenced at www.cplusplus.com/reference/string/string/. This manual will cover just a few common functions that you will used in your project. Feel free to check out the tutorial at www.geeksforgeeks.org/c-string-class-and-its-applications/ for more on *string* use.

When working with *strings*, we often want to check if one *string* is equivalent to another. This can be accomplished using the compare() function (Fig. 6). Given some *string* str1, compare() takes another *string* str2 as an argument to check if str1 is the same as str2. If so, the function returns the value 0, otherwise it returns a non-zero value.

Figure 6: Compare the Equivalency of Two Strings

```
int found = str1.compare( str2 );
```

Alternatively, we may not want to verify the equivalency of two *strings,* but instead the presence of one *string* within another. Fig. 7 demonstrates how this can be implemented using *string's* find() function. A second *string* str2 is inputted as the argument, and find() quires str1 to see if it contains str2. If str2 is found, the index of the first occurrence of str2 in str1 is returned. The index value will be returned as an unsigned integer type **size_t**. For that reason, to indicate that str2 was not found, the value **string::npos** is returned to indicate some out-of-bound index.

Figure 7: Verify if str1 Contains str2

```
size_t index = str1.find( str2 );
```

Based on the composition of str1, we may decide to replace a segment of the *string* with another substring. To implement, use the replace() function (Fig. 8) on str1 with these arguments: the size_t index of the first character in str1 to replace, the size_t total number of characters in str1 to replace, and a *string* str2 for replacement. This function does not return a value.

Figure 8: Replace Substring of str1 with str2

```
str1.replace( 2, 7, str2 );
```

What if you want retrieve a substring of str1 instead of replacing it with another *string*? Fig. 9 shows how this can be done using the substr() function. Enter in the size_t arguments of the substring's first index and the total number of characters in the substring. The return value is a *string* object of the requested substring.

Figure 9: Retrieve Substring

```
str1.substr( 2, 7 );
```

The last function I want to discuss is not a member function of the *string* class, but it can be used to modify the characters of a *string* object. This function is called tolower() (Fig. 10). It takes a char variable as an argument, and returns an int value of the lowercase equivalent for the char argument. To obtain an actual char variable, cast the returned value as char following the code in Figure 10. Since the *string* class does not have a built-in lowercase conversion function, tolower() will be crucial if you want to convert the text of a ***document*** to lowercase before conducting text analysis.

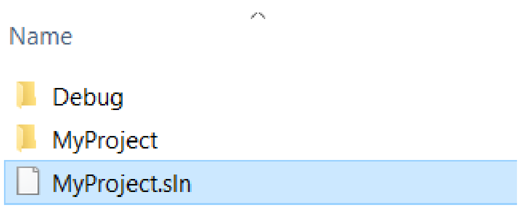
Figure 10: Convert Entire String to Lowercase

```
string str1 = "E.T. phone home.";
string str2 = "";
for ( size_t i = 0; i < str1.size(); i++ )
{
    str2 = str2 + char( tolower( str1[i] ) );
}
```

Building Program:

You're ready to add the next class ***sample***, to your program. The ***sample*** class represents an instance of data; meaning a row of data in a database (think of an Excel sheet). Each ***sample*** contains a ***document*** object and a pointer to a string array. This string array represents the data fields of information about the text stored in the ***document*** object. A data field might be the year the text written. Follow the instructions below to start constructing the sample.h file.

1. Open up Visual Studio using the icon or the start menu .
2. Go to menu bar and click: File → Open → Project/Solution....
3. In the Open Project window, locate your project's saved program. Enter its folder and select MyProject.sln (or the solution file of the name you chose for your project). Click Open.


Name
Debug
MyProject
MyProject.sln

4. Create a new source file. Right click on Source Files in Solution Explorer, then select: Add →New Item…
5. In the Add New Item - Document window, select Visual C++ → C++ File (.cpp). Title the file sample.cpp and click Add. Then repeat, but select the Header File (.h) option to create sample.h.

6. Given your practice with creating a class, try to build sample.h from just a description of its contents. Refer to syntax from document.h when needed. Check Appendix B-2 for an example solution of the ***sample*** class declaration.

   sample.h File:
   Public:
   - Default constructor
   - Constructor that takes the parameter int nFields
   - Constructor that the parameters *string* s and int nFields
   - Destructor
   - Function getNumOfFields() that takes no parameters and returns an int
   - Const function getField() that takes an int parameter index and returns a *string* object
   - Const function getText() that takes no parameter and returns a *string* object
   - Function setField() that takes an int parameter index and *string* parameter data. Nothing returned.
   - Function setText() that takes a *string* parameter data. Nothing returned.
   - Function emptySample() that takes no parameters and returns nothing

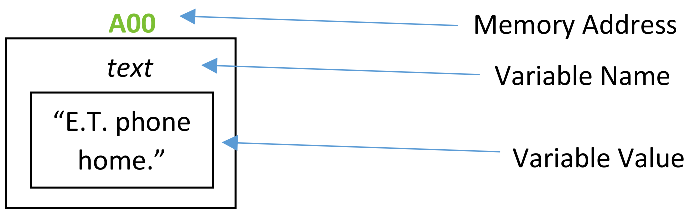
   Private:
   - ***document*** object called text
   - int value called numFields
   - *string* pointer called datafields. It should take the form: std::string * datafields.

7. Save your work.

Pointers

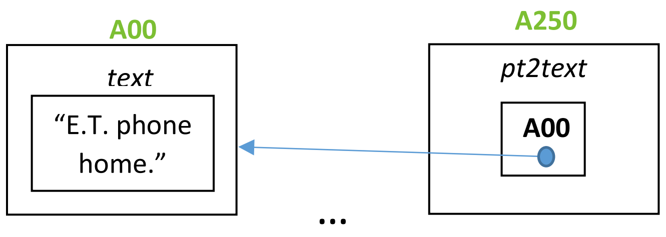
In the Arrays and Strings section you created the beginnings of the *sample* class. We will continue to develop this class as you learn more about the functionality of pointers. Going back to the sample.h file, each *sample* object is defined as having a private member std::string* datafields. Quickly glancing at this variable, you might read it as a string object named datafields. However, the ***** of string * alters the role of the variable, turning it into a string pointer. A **pointer** is a variable that stores the memory address of, and therefore essentially points to, another variable. Consider the string member variable called text from the ***document*** class. When that variable text is created, your program will use a space in the available memory to create and store the string. The location of text will have a unique address in order to make it easily accessible whenever text is read from or written to in your program (Fig 1).

Figure 1: Memory Allocation of Variable *text*



Now if you wanted a pointer called pt2text to point at the variable text, see Figure 2 for a visualization of what the pointer does.

Figure 2: Pointer pt2text's Role in Memory



Like any other variable, a pointer must be declared and initialized. Figure 3 shows the acceptable format for both of these actions.

Figure 3: Pointer Declaration and Initialization

```
string text ="E.T. phone home."

//Declaration:

String *pt2text;
string *pt2quote;
string *pt2tweet;

//Initialization:
pt2text = &text;
pt2quote = p2text;
p2tweet = new string {"movie night #spielberg" };
```

Declaration follows the pattern: type * name; where type indicates the only value type to which the pointer can point.

The Address-of operator, &, returns the memory address of a variable. Given Fig. 2, &text would store a00 in pt2text.

The C++ language is lenient in regards to the location of the * during declaration. All three declaration formats listed in Figure 3 are accepted; however int *pt2tweet is considered best practice for declaring a pointer. The reasoning for this, is if you decided to declare more than one pointer on one line, each pointer requires its own *. Tethering * to the pointer's name decreases your chances of forgetting this rule.

So, `int* pt1, pt2, p3;` would be considered incorrect. The proper form for multiple pointer declarations would be `int *pt1, *pt2, *p3;` .

The best form for * placement changes when returning a pointer from a function. In that case, `int*` is preferred to make it clear that an int pointer is being returned from the function.

When initializing a pointer, remember that a pointer stores a memory address. There are three ways in which a pointer can be initialized with a memory address (see Fig 3). First, it can use the & operator to access the memory location of a specified variable. Second, you can initialize a pointer with another pointer. This simply indicates that the initialized pointer should store the same memory address as the other pointer. Third, a pointer can be set using the **new** operator. The new operator takes the format of: new type{}, where a new variable of the given type is generated. The memory address of this newly constructed variable is stored into the pointer.

In addition to declaration and initialization, pointers can be dereferenced to obtain the value to which the pointer is pointing at. The dereference operator is also a *. To use this operator see Fig 4. The code in Figure 4 should print out “Sample Text.”

Figure 4: Dereferencing a Pointer

```
string text = “Sample Text.”;
string *pt2text = &text;
cout << *pt2text << endl;
```

You can also have pointers point to an array (Fig. 5).

Figure 5: Pointer to array

```
//Method 1:
int arr[3] = {1, 2, 3};
int*pt= arr;

//Method 2:
int *pt = new int[3];
```

The functionality of pointers can grow through the use of a double pointer (**). Four situations can be represented by a double pointer:

1. Pointer to Pointer to Value

```
int *pt1 = new int{7};
int *pt2 = &pt1;
```

2. Pointer to Pointer to Array

```
int *pt1 = new int[2]{ 1, 7 };
int *pt2 = &pt1;
```

3. Pointer to Array of Pointers to Values

```
int *pt = new int*[2];
for( int i = 0; i < 2; i++ }
{
   pt[i] = new int{2};
}
```

4. Pointer to Array of Pointers to Arrays

```
int *pt = new int*[2];
for( int i = 0; i < 2; i++ }
{
   pt[i] = new int*[2];
   for( int j = 0; j < 2; j++ }
  {
     pt[i][j] = new int{2};
  }
}
```

When you are done using a pointer, you can free up its memory space by using the delete keyword as seen in Figure 6.

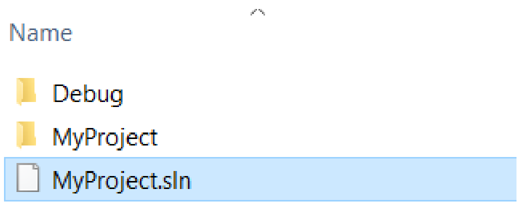
Figure 6: Deleting Pointers

```
//Delete single pointer (*):
delete ptr;

//Delete pointer to an array :
delete [] ptrArr;

//Delete double pointer(**):
// (pointer to pointer to value)
delete *pt;
delete pt;
```

If pointer is pointing to an array of pointers, you need to call delete on each pointer in the array before deleting the exterior pointer.

<u>Building Program:</u>
Keeping the syntax of pointers in mind, create the implementation (sample.cpp file) of the ***sample*** class.

1. Open up Visual Studio using the icon or the start menu .
2. Go to menu bar and click: File → Open → Project/Solution....
3. In the Open Project window, locate your project's saved program. Enter its folder and select



Name
Debug
MyProject
MyProject.sln

4. To construct the sample.cpp file follow the implementation instructions for each function. Check Appendix B-3 for an example solution of sample.cpp.
   <u>sample.h File:</u>
   - **Default Constructor:** sets text variable to an empty *string*, numFields to 1, and datafields to a **new** *string* array of size 1.
   - **Single Parameter Constructor:** sets text variable to an empty *string*, numFields to the int parameter, and datafields to a **new** *string* array of the size numFields.
   - **Double Parameter Constructor:** sets text variable to the *string* parameter, numFields to the int parameter, and datafields to a **new** *string* array of the size numFields.
   - **Destructor:** calls emptySample().
   - **getNumOfFields():** returns copy of numFields.
   - **getField():** checks if int parameter is within the range of numFields, then returns copy of requested index from *string* array datafields.
   - getText(): returns copy of *string* text.

    - setField():checks if int parameter is within the range of numFields, then sets the requested index of datafields with the *string* parameter.
    - setText(): sets string text with value of *string* parameter.
    - emptySample(): calls delete on the datafields array.
5. Save your work and run program for debugging.

Introduction to Text Analysis and Udat

When constructing the *document* and *sample* classes earlier, you were coding an object representation of a text sample and data instance used for **text analysis**. With the basics of C++ covered, it's time to consider this specialty topic.

What is text analysis exactly? It is the study of a set of text with the purpose of locating significant patterns in the text. These patterns are determined after reviewing quantitative statistics about the text. Quantitative statistics can include the frequency of topic-specific words, the frequency of words used together or individually, the total number of words in a given text document, etc. Each of these statistics is also referred to as a **text feature**.

Beyond reviewing these numerical features, we may want to see if the computer can recognize patterns associated with classes, categories, of the text set. In your project, text documents will be classified according to the year or period the text was released. Looking at poems by Shel Silverstein, all of his poems published in 1974 would be categorized in one class, and all of his poems published in 1981 would be categorized in another class (see Fig. 1).

Figure 1: Classes—Silverstein Poem Titles Categorized By Year

Title	Class
Where the Sidewalk Ends	1974
Eighteen Flavors	1974
Stone Telling	1974
How Many, How Much	1981
A light in the Attic	1981
Moon Catchn' Net	1981

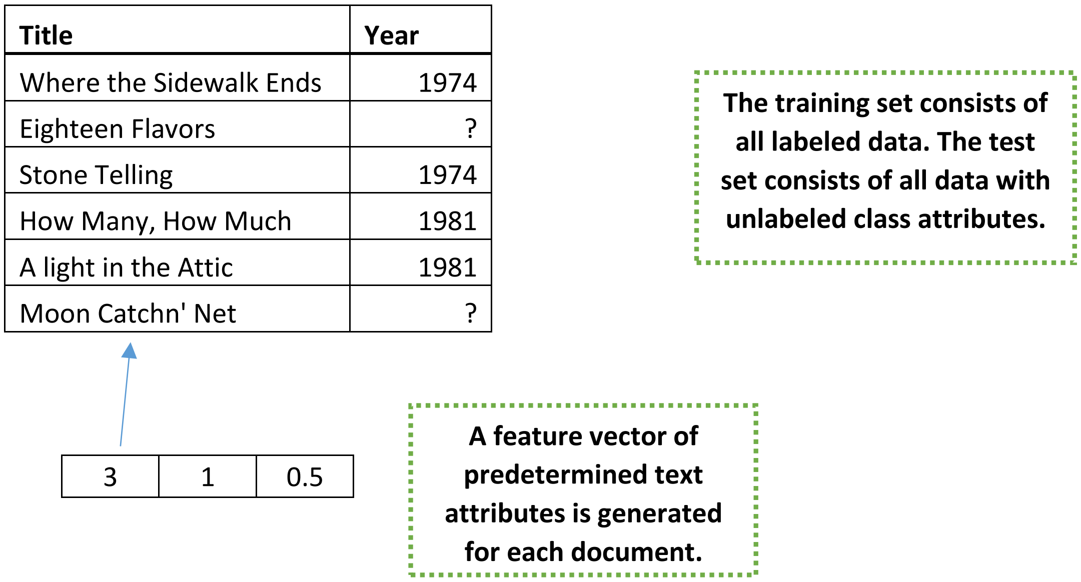
Class Attribute

Text classification takes pattern recognition one step further by removing the class labels of previously categorized text, in order to predict the class of the text based on text features (patterns) alone.

To perform text analysis and classification, the text analysis tool Udat, created by Professor Lior Shamir, will be used to implement your experiment. When running Udat, the data will be split into two sets (Fig. 2), a training set and a test set. The **training set** uses labeled text samples, or **documents**, to teach the computer what the average features for a class looks like. The **test set** removes the class attribute value in order to test how accurately the classifier (Udat) is able to predict a document's class based on what it has learned. To compare documents, Udat will extract a numeric array of text features, known as a

feature vector, from each training sample to find patterns in the training set and use this as a comparison mechanism between data in the training and test sets.

Figure 2: Splitting Data into Training and Test Sets



Title	Year
Where the Sidewalk Ends	1974
Eighteen Flavors	?
Stone Telling	1974
How Many, How Much	1981
A light in the Attic	1981
Moon Catchn' Net	?

3	1	0.5

The training set consists of all labeled data. The test set consists of all data with unlabeled class attributes.
A feature vector of predetermined text attributes is generated for each document.

The resulting feature vectors are applied to a machine learning algorithm to predict the class label of documents in the test set. The machine learning algorithm used by Udat is the Weighted Nearest Neighbor (WNN) algorithm that measures the weighted Euclidian distance of a test document's feature vector from each member in a class's training set. The class with the minimum weighted distance to the test document is considered the predicted class of the test document.

If you curious as to how this is exactly computed, the equation for WNN is:

$$d_{x,c} = \min_{t\in T_c}\left[\sum_{f=1}^{|x|} w_f (x_f - t_f)^2\right].$$

In this equation, $d_{x,c}$ represents the distance between a test document's feature vector x and a given class c. T_c is the training set for the class, *t* is the feature vector for T_c, |x| is the total number of features in vector x, *f* is a specific feature, w_f is *f*'s Fisher's score, x_f is f's value of from a test set vector x, and t_f is the value of *f* from a training set vector.

After the WNN algorithm and any additional calculations are executed, Udat produce results indicating how well it predicted the class of documents in the test set, as well as any additional information pertaining to the similarities of features vectors and specific features across classes. If you are interested in learning more about how Udat works, read the paper:
https://scfbm.biomedcentral.com/articles/10.1186/1751-0473-3-13
by Lior Shamir, Nikita Orlov, D Mark Eckley, Tomasz Macura, and Josiah Johnston about a very similar image analysis tool called WndChrm.

Before learning how to run Udat, you will need to first download the software from:
http://vfacstaff.ltu.edu/lshamir/downloads/udat/

Please read the text_analysis_protocol Word file on Blackboard to answer any additional Udat download questions and future usage questions. Move on to the next section to start brainstorming the subject of your research project.

Choosing your Data

Before you begin investigating what data to analyze, let's explain the goal of this project. You are going to explore the patterns found in text using a set of collected data by a single *source* (creator). By "*source*," this could be your favorite author, journalist, etc. You will be analyzing work by this *source* over two different years. The purpose of this is to discover if there are any distinguishing features found in their work based on the year of its creation. When choosing your data, look for texts with at least a decade gap between them. Note that you may also choose to compare texts across two different periods as opposed to two different years. For example, you might compare an author's novels from the 1910s to their novels from the 1930s. Additionally, think about what each document of data will look like. If you have a music artist that you are interested in, you can compare lyrics of songs by that artist across different albums, or you may be interested in just comparing the choruses of songs across their albums. How you define and split up your data is entirely up to you.

This experiment will be focused only one *source* in order to increase the ease of data collection and understanding of text mining. At the end of this experiment, there will be a practice activity where you can take what you've learned and apply it to a comparison of two or more *sources*. Until then, you're first exercise is to think about what data you would like to analyze for your project. Below are some brainstorming questions to aid you in your decision, but if you already know what data you're interested in, feel free to move on to the next section.

BRAINSTORMING:

- What are your hobbies? Interests?
- Are you a fan of certain music, movies, literature, social media, etc.?
- Is there any text associated with your interests?
    - Lyrics, screenplays, novels, poems, blogs, articles, tweets, etc.
- What section of this text do you want to analyze?
    - A whole song or the chorus of a song?
    - A movie quote or a movie title?
- Do you have a favorite *source* for this subject of text?
    - Favorite musician, screenwriter, author, poet, blogger, etc.
    - Your *source* doesn't necessarily have to be the creator of the text.
      For example, you can look at screenplays interpreted by your favorite film director.
- It might help to think about the genre of the text, and if the *source* is from the present day or the past.
- If you need some inspiration:
    - What song/artist is most played on your iPod or phone?
    - What movies or TV shows do you own, have watched recently, or have watched most often?
    - What books do you own, have read recently, or have read most often?
    - Take a look around your room. Do you see anything of interest that is based on a text source?
- Still unsure? Browse the web!
    - Research a *source* you would enjoy analyzing.
    - It's also a handy way to determine what a sample of text will look like.

Hopefully, you were able to find some inspiration from the brainstorming guide. If you are still unsure of your project idea, that's okay. Take some time to think it over and come back when you feel ready. As you go about constructing your own experiment, I will explain each step of the process using a research experiment of my own: comparing samples of literary work by William Butler Yeats (*source*) from pre-1990 to post-1910.

Collecting Data

Now that you've determined a data source to analyze, it's time to collect your data. The data collection method is entirely up to you. There are certain websites that provide public access to data that you may use. An example of such a site, is Project Gutenberg[2], where I chose to collect literary works by Yeats.

As you are collecting data, it is important to note that you will need a minimum of 50 data samples (document objects) for each time year/period considered. Each sample should be a10 characters or more. Don't work if you can't find 50+ data samples by your *source*. If you are able to locate a larger text sample, it can be split into smaller samples. I analyzed only 5 collections of works for each time period, but divided up text to create 4-line sample during preprocessing. We will go over the code for this in the

[2] www.gutenberg.org

File Input/Output section. After your data is gathered, it should be saved in .txt files, if it isn't already, before continuing on with the project. Make sure you have a basic text editor, like Notepad, to view and/or edit the files later on.

File Input and Output

Your data is collected and saved, but in order to use it for text analysis, you will need to be able to access and modify your data using C++. This is done using **file streams** of file input/output (I/O).

Every time you call a print statement, **cout**, you are sending screen output to a file stream. Think of a file stream as a channel between your program and a file that allows the free flow of bytes of data between the two entities. File streams give programmers access to read and write data *sequentially*. The term sequentially is key. Imagine you had a list of five things to do today. You've completed three of those things, but forgot what the last two tasks are. Unlike arrays, with streams you are not able to directly call access points (indices) to data. Instead, you must read through the first three items on your to-do list again before reaching those last two tasks. Reading a file stream works in a similar manner.

While cout required you to include the *<iostream>* class library header, basic reading and writing to a file in C++ requires the inclusion of both the *<iostream>* and the *<fstream>* headers. The addition of fstream, provides three stream classes for file input and output:

- ofstream: used to create and write to files
- ifstream: used to read files.
- fstream: used to create, write, and read files.

To open a file or create a new file, one of the two methods in Fig. 1 can be used. When opening a file, the stream object chosen should match the purpose for the opening file; whether it be to read or write data, or both.

Figure 1: Basic Opening of a Text File

```
//Method 1:
ifstream file;
file .open( "filename.txt" );


//Method 2:
ifstream file( "filename.txt" );
```

Stream object named file

Note:
You can use a
fstream object
here as well.

Opening a file follows the syntax of open(filename or absolute file path, mode); where mode applies special attribute to the stream object. To learn more about mode and file input/output in general, visit www.cplusplus.com/doc/tutorials/files/. Once a file has been opened, it is often good practice to check if the file was properly opened using the .is_open() function (Fig. 2).

Figure 2: Boolean check of Open Status

```
ifstream file( “filename.txt” );
if( file.is_open() )
{
  //Okay to continue processing
}
else
{
  cout << “File failed to open.” << endl;
}
```

Use print statement to notify user that an error has occurred.

The last function you need to know before performing reading and writing tasks is how to close your opened file. Adding the statement shown in Fig. 3 will immediately close your file and open up the stream object’s memory for other uses.

Figure 3: Close File

```
ifstream file( “filename.txt” );
if( file.is_open() )
{
  //Read or write
  file.close();
}
else
{
  cout << “File failed to open.” << endl;
}
```

With opening and closing a file established, you are now ready for the next step: writing to and reading data from a file. To write data with ofstream, the format is similar to *cout*, except instead of applying the insertion operator (<<) to *cout*, it is applied to the stream object (Fig 4).

Figure 4: Writing to a File

```
ofstream theFile( "filename.txt" );
if( theFile.is_open() )
{
  theFile << "Write to file." << endl;
  theFile << "Use the insertion operator.";

  theFile.close();
}
else
{
  cout << "File failed to open." << endl;
}
```

If a file does not already exist, it will be created before any writing begins.

Note:
You can use a fstream object here as well.

In reading a file with ifstream (Fig. 5), the insertion operator (>>) of *cin*, keyboard input, can be used to read a file word by word, breaking at each occurrence of a space. To read a whole line of text, use the getline() function.

Figure 5: Reading a File

```
ifstream theFile( "filename.txt" );
if( theFile.is_open() )
{
  string input;
  theFile >> input;
  getline ( theFile, input );

  while( getline( theFile, input ) )
  {
    cout << input << endl;
  }

  theFile.close();
}
else
{
  cout << "File failed to open." << endl;
}
```

Individual call to read word, then read line.

To read an entire file, use a while loop to call getline() until the file ends.

Preprocessing

Data preprocessing is the modification of your data into a format more acceptable for analysis. It can take many forms, such as: removing white space, converting to lowercase, removing punctuation, removing hyperlinks, etc. The type of preprocessing required is entirely dependent on the data type, the analysis tools, and the goals of your research. In this project, none or very little preprocessing is needed. Take a moment to review your data. Are there any elements in your data that would skew an analysis of its features or are simply useless in your experiment?

In preprocessing my data on Yeats I found four things I wanted to change:

1. Convert all text to lowercase. The data I had used uppercase at random moments, and was not reflective of the original work.
2. Remove character names that were repeated multiple times within the text. This is not restricted to just names, but can be applied to other keywords used often in your text. The purpose of the removal is to avoid Udat using these keywords as identifiers instead, of relying on other inherit elements of the text.
3. Remove any spaces that prefixed the beginning of the line to normalize the indention scheme of across all the documents.
4. Split the large text files into smaller text files containing 4 consecutive lines of text. In this manner you can increase the size of your data set, and test if Udat can correctly classify a document when looking at just a small sample size.

Note: when saving you text files after preprocessing, follow this format to ensure your code works with final class that you will create, the ***database*** class:

C:\PathToProjectFolder\classID_fileID.txt

OR (if identifying object by collection as well)

C:\PathToProjectFolder\classID_collectionID_fileID.txt

Cleaning Files

With preprocessing, you files may require different cleaning task text files that I used in my experiment. However, in case you have similar tasks, I will describe what string functions you will need to implement them. To convert each line to lowercase you will use the tolower() function. To remove any prefixing spaces you can use the substr() function to extract the segment of the line that contains the actual text. In removing names, you would need to combine the find() and replace() functions. Also, the compare function will come in handy when you want to initially recognize if a string is empty or not. If it is, you don't need to do any cleaning.

When performing your cleaning tasks, you can define them in individual functions or in one over-arching function.

I implement my cleaning tasks in one function, and provide the function with the parameters:

- *string* of the file's class
- int of the file's id
- *string* of the file's path

- *string* pointer of the array of names to remove
- int length of the array of names

Given this structure, I stored all the arrays of names to remove in the main function, and simply sent a pointer to the relevant array every time I cleaned a file.

Refer to preprocessFiles.sln file from Blackboard on how I implemented my cleanFile() function. Use it as a base, and alter what you need to perform the cleaning tasks necessary for your project and your data.

Splitting Large Text Files into Smaller Documents

If your data needs to be split into smaller samples, you can implement this using the compare() function to indicate when a break in the lines has occurred. To execute the split, a separate function, splitFile(), was called for each text file. Refer to preprocessFiles.sln file from Blackboard for an example solution of its implementation. Once again, your code may be different, especially if you did not divide the text files by both class and collection, as I did.

Preprocess your files, and move on to the next section when you are complete.

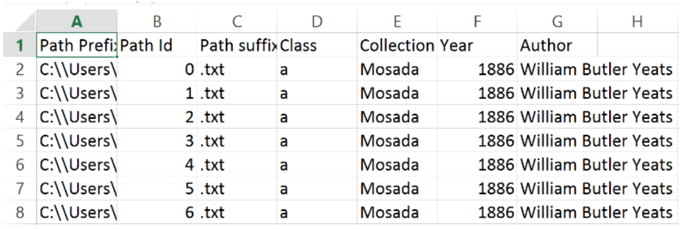
Creating CSV File

For this project, you will need to create a **CSV file** of information about each text sample (document). A CSV (comma-separated values) file is a text file that stores data in a tabular format, allowing it to be used as a spreadsheet/database in Excel. Each line in a CSV file represents a **data instance**, a collection of information on one item or person. This information is categorized into fields. The CSV file separates each field with a comma, hence comma-separated values. For this reason, any text store in your CSV should not contain a comma, because your program will be using commas as identifiers between fields.

Considering the tabular format of the data, each data instance can be viewed as an array (Fig 1).

Figure 1: Each Data Instance is an Array

CSV File of Data:



	A	B	C	D	E	F	G	H
1	Path Prefi	Path Id	Path suffi	Class	Collection	Year	Author	
2	C:\\Users\	0	.txt	a	Mosada	1886	William Butler Yeats	
3	C:\\Users\	1	.txt	a	Mosada	1886	William Butler Yeats	
4	C:\\Users\	2	.txt	a	Mosada	1886	William Butler Yeats	
5	C:\\Users\	3	.txt	a	Mosada	1886	William Butler Yeats	
6	C:\\Users\	4	.txt	a	Mosada	1886	William Butler Yeats	
7	C:\\Users\	5	.txt	a	Mosada	1886	William Butler Yeats	
8	C:\\Users\	6	.txt	a	Mosada	1886	William Butler Yeats	

Array for Data Instance #2:

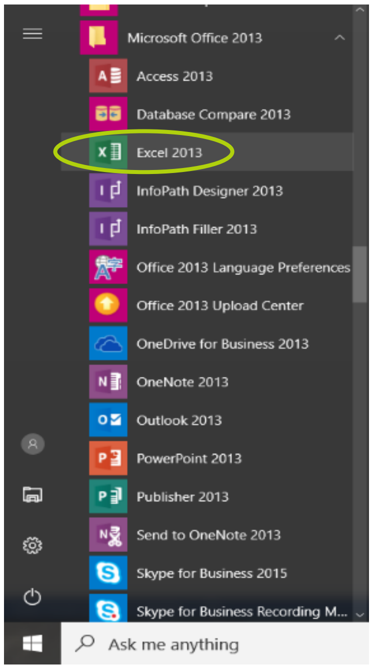
0	1	2	3	4	5	6
C:\\PathPrefix	1	.txt	a	Mosada	1886	William Butler Yeats

The CSV file that you create will contain a minimum of 4 fields: the path prefix of the text file, the unique id of the text file, the .txt suffix of the file, and the class name of the file. Any additional fields can be added for further analysis. An inclusion of the year the text was written is helpful when comparing text across time periods; allowing you would to refer to the exact year of publication. Also, including an author field is beneficial for the extra segment of the project where you analyze data from two different *sources*.

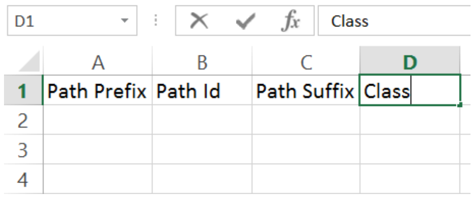
To create your CSV file, the most straightforward approach is to build it manually in Excel. Follow the instructions below to create your file. Keep in mind that this CSV file will be linked to your C++ program with a new class called *database*. For this reason, the minimum four fields needed in your file should be created in the same order as the instructions to ensure your program runs correctly.

<u>Building CSV:</u>

1. Open up from your computer's version of Excel. This can be done by either clicking the icon or finding Excel 20 _ _ under Microsoft Office in the start menu.



2. Type in the four required fields exactly as shown in the image below.



	A	B	C	D
1	Path Prefix	Path Id	Path Suffix	Class
2				
3				
4				

3. Next begin to fill in the CSV file. Add the Path Prefix to cell b1. This Path Prefix will be the same for all your text files, with a minor change for each class ID (and collection ID if relevant). If you recall from the preprocessing exercise, files should be saved in the format of:

   $$\underbrace{\text{C:}\backslash\text{PathToProjectFolder}\backslash\text{classID\_}}_{\text{Path Prefix}}\text{fileID.txt}$$

   (In my project, I wanted to also identify which collection a document derived from, so I used the format:

   $$\underbrace{\text{C:}\backslash\text{PathToProjectFolder}\backslash\text{classID\_collectionID\_}}_{\text{Path Prefix}}\text{fileID.txt}$$

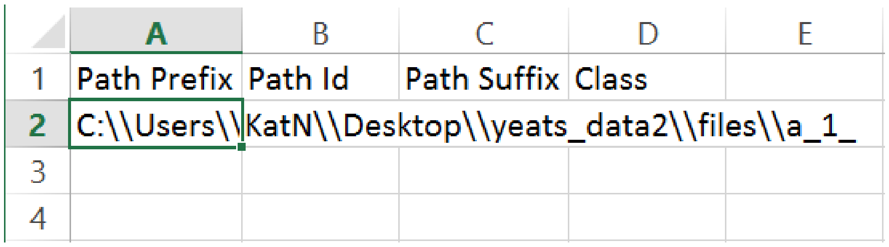
)

   Important: when writing the path to a file in the CSV document, you need to replace every \ in the path with a \\. This ensures the path is in the correct format for your C++ program to open files with.

   So, based on the number text files you have in each class (and also collection), you will use Excel's drag-and-copy feature to quickly write the prefix for each file of the class.

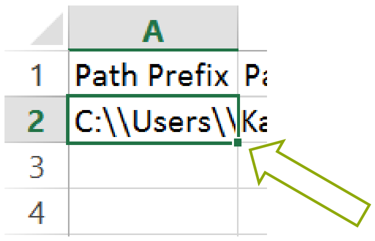
   To use this feature:

   a. Select the cell you would like to copy.



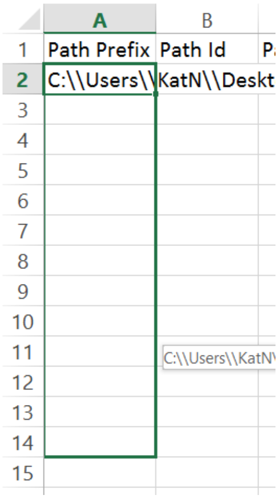
	A	B	C	D	E
1	Path Prefix	Path Id	Path Suffix	Class	
2	C:\\Users\\KatN\\Desktop\\yeats_data2\\files\\a_1_				
3					
4					

   b. Move your mouse to the tiny green box on the lower right-hand corner of the cell until you mouse changes from a white to black plus sign.

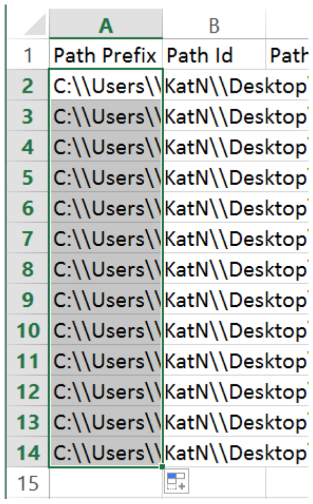


	A
1	Path Prefix
2	C:\\Users\\
3	
4	

c. Click and drag the mouse down the column for as many number of cells as text files you have for that Path Prefix. Release when you have reached that number.

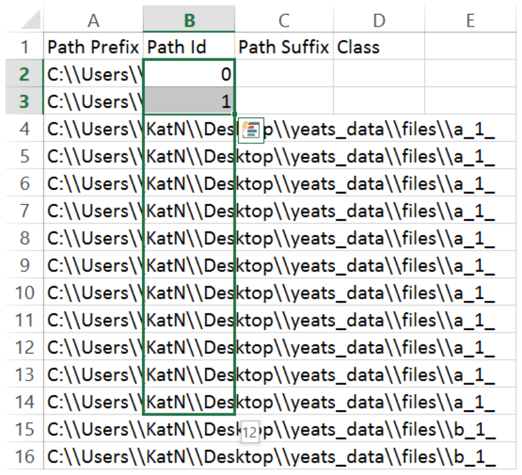


	A	B	
1	Path Prefix	Path Id	P
2	C:\\Users\\KatN\\Deskt		
3			
4			
5			
6			
7			
8			
9			
10			
11		C:\\Users\\KatN\	
12			
13			
14			
15			



	A	B	
1	Path Prefix	Path Id	Path
2	C:\\Users\\KatN\\Desktop		
3	C:\\Users\\KatN\\Desktop		
4	C:\\Users\\KatN\\Desktop		
5	C:\\Users\\KatN\\Desktop		
6	C:\\Users\\KatN\\Desktop		
7	C:\\Users\\KatN\\Desktop		
8	C:\\Users\\KatN\\Desktop		
9	C:\\Users\\KatN\\Desktop		
10	C:\\Users\\KatN\\Desktop		
11	C:\\Users\\KatN\\Desktop		
12	C:\\Users\\KatN\\Desktop		
13	C:\\Users\\KatN\\Desktop		
14	C:\\Users\\KatN\\Desktop		
15			

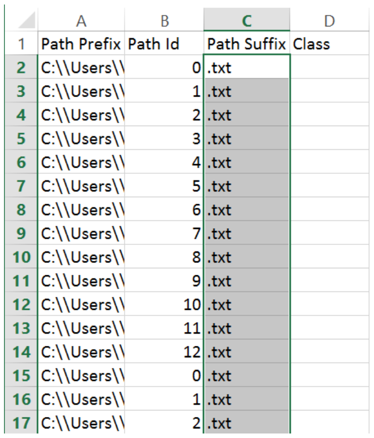
4. Repeat step 3 for the Path Prefix of your second class. (Repeat for each collection if necessary).
5. Add the Path Id. Simply write 0 in cell b2 and 1 in cell b3. Select both cells b2 and b3, click the lower right-hand box, and drag your mouse downwards. Excel will generate the next whole for you. Drag until you reached the highest Path Id for files in your first class.



	A	B	C	D	E
1	Path Prefix	Path Id	Path Suffix	Class	
2	C:\\Users\\	0			
3	C:\\Users\\	1			
4	C:\\Users\\KatN\\Desktop\\yeats_data\\files\\a_1_				
5	C:\\Users\\KatN\\Desktop\\yeats_data\\files\\a_1_				
6	C:\\Users\\KatN\\Desktop\\yeats_data\\files\\a_1_				
7	C:\\Users\\KatN\\Desktop\\yeats_data\\files\\a_1_				
8	C:\\Users\\KatN\\Desktop\\yeats_data\\files\\a_1_				
9	C:\\Users\\KatN\\Desktop\\yeats_data\\files\\a_1_				
10	C:\\Users\\KatN\\Desktop\\yeats_data\\files\\a_1_				
11	C:\\Users\\KatN\\Desktop\\yeats_data\\files\\a_1_				
12	C:\\Users\\KatN\\Desktop\\yeats_data\\files\\a_1_				
13	C:\\Users\\KatN\\Desktop\\yeats_data\\files\\a_1_				
14	C:\\Users\\KatN\\Desktop\\yeats_data\\files\\a_1_				
15	C:\\Users\\KatN\\Desk 12 p\\yeats_data\\files\\b_1_				
16	C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_1_				

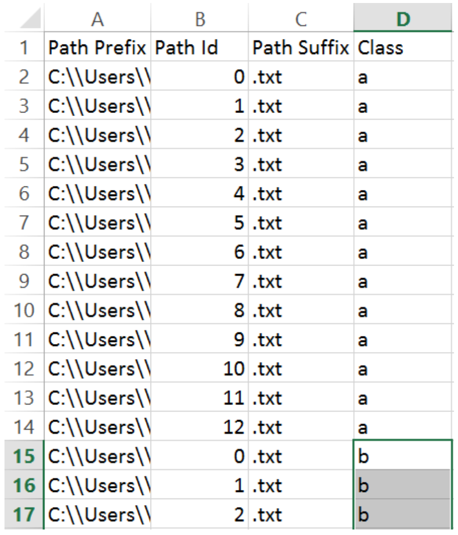
Repeat this step in the two cells below your last generated id in order add the ids for your next class/collection.

6. Add the .txt suffix to cell c2. This is the same for all files, so just drag-and-copy until the last data instance is reached.



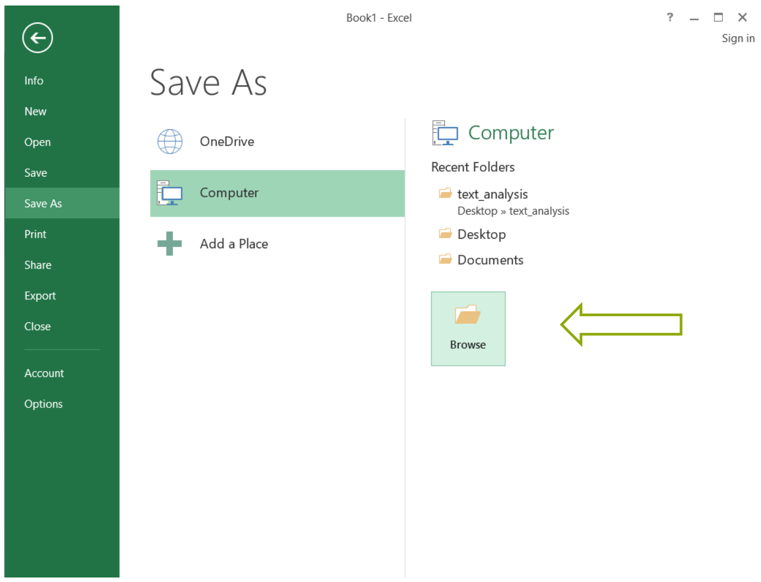
	A	B	C	D
1	Path Prefix	Path Id	Path Suffix	Class
2	C:\\Users\\	0	.txt	
3	C:\\Users\\	1	.txt	
4	C:\\Users\\	2	.txt	
5	C:\\Users\\	3	.txt	
6	C:\\Users\\	4	.txt	
7	C:\\Users\\	5	.txt	
8	C:\\Users\\	6	.txt	
9	C:\\Users\\	7	.txt	
10	C:\\Users\\	8	.txt	
11	C:\\Users\\	9	.txt	
12	C:\\Users\\	10	.txt	
13	C:\\Users\\	11	.txt	
14	C:\\Users\\	12	.txt	
15	C:\\Users\\	0	.txt	
16	C:\\Users\\	1	.txt	
17	C:\\Users\\	2	.txt	

7. Use the drag-and-copy feature to add the class name of each text file.

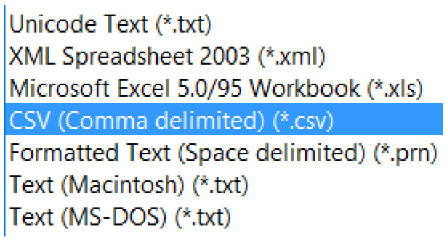


	A	B	C	D
1	Path Prefix	Path Id	Path Suffix	Class
2	C:\\Users\\	0	.txt	a
3	C:\\Users\\	1	.txt	a
4	C:\\Users\\	2	.txt	a
5	C:\\Users\\	3	.txt	a
6	C:\\Users\\	4	.txt	a
7	C:\\Users\\	5	.txt	a
8	C:\\Users\\	6	.txt	a
9	C:\\Users\\	7	.txt	a
10	C:\\Users\\	8	.txt	a
11	C:\\Users\\	9	.txt	a
12	C:\\Users\\	10	.txt	a
13	C:\\Users\\	11	.txt	a
14	C:\\Users\\	12	.txt	a
15	C:\\Users\\	0	.txt	b
16	C:\\Users\\	1	.txt	b
17	C:\\Users\\	2	.txt	b

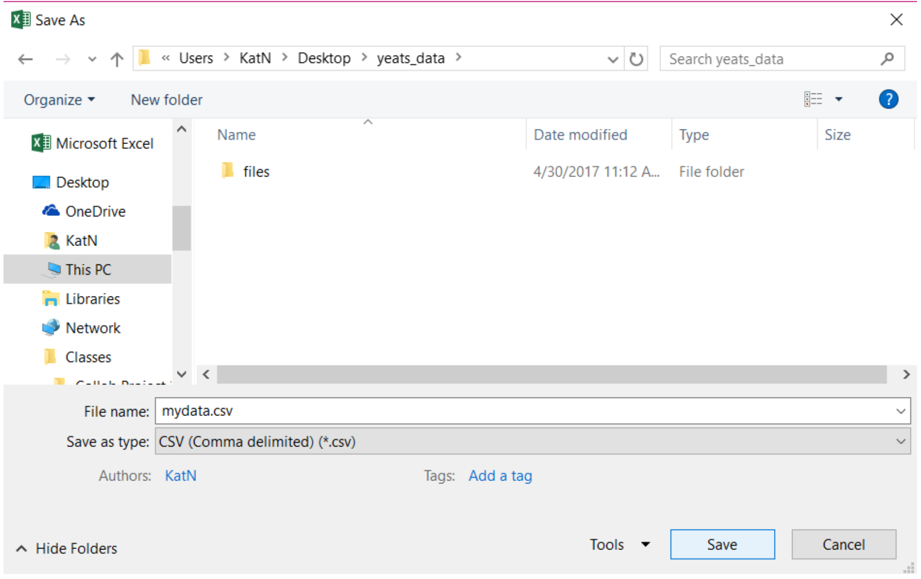
8. If you have any additional fields, use drag-and-copy to generate what you data that you can.
9. Save mydata.csv by going to:
    a. File → Save
       OR
    b. File→ Save As → Browse.



Navigate to your project's folder, and alter the Save as Type to CSV (Comma delimited) (*.csv).


Unicode Text (*.txt)
XML Spreadsheet 2003 (*.xml)
Microsoft Excel 5.0/95 Workbook (*.xls)
CSV (Comma delimited) (*.csv)
Formatted Text (Space delimited) (*.prn)
Text (Macintosh) (*.txt)
Text (MS-DOS) (*.txt)

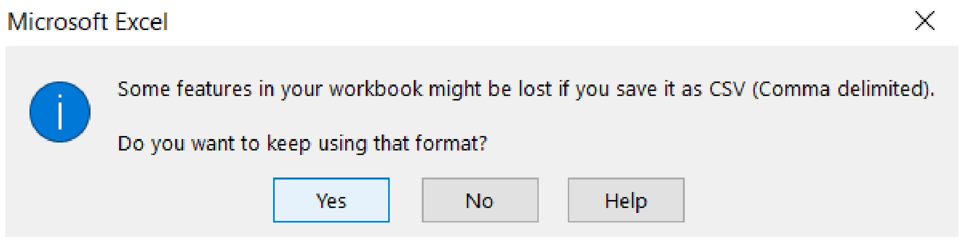
Title the file mydata.csv, and then click Save.



File name: mydata.csv

Save as type: CSV (Comma delimited) (*.csv)

Authors: KatN   Tags: Add a tag


Microsoft Excel

Some features in your workbook might be lost if you save it as CSV (Comma delimited).

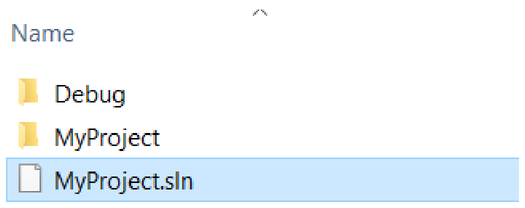
Do you want to keep using that format?

Linking Program to Udat

In building your document and sample class, collecting and preprocessing your data, and creating a CSV file for your data, you have set up the foundations for the link between your C++ program and Udat. The final step is create class called database. A database object will represent an array of samples and will be built from the CSV file that you have made. This abstract version of a database connects to Udat through a function called buildAsociation() that generates a single text file linking each text fill with its class. This file will be used as input when you run Udat on you text samples.

<u>Building Program:</u>
Add ***database*** class to your program.

1. Open up Visual Studio using the icon or the start menu .
2. Go to menu bar and click: File → Open → Project/Solution....
3. In the Open Project window, locate your project's saved program. Enter its folder and select MyProject.sln (or the solution file of the name you chose for your project). Click Open.


Name

Debug
MyProject
MyProject.sln

4. Create a new source file. Right click on Source Files in Solution Explorer, then select: Add →New Item...
5. In the Add New Item - Document window, select Visual C++ → C++ File (.cpp). Title the file database.cpp and click Add. Then repeat, but select the Header File (.h) option to create database.h.
6. Given your practice with creating a class, try to build database.h from just a description of its contents. Refer to syntax from document.h when needed. Check Appendix C-1 for an example solution of the ***database*** class declaration.

database.h File:

Public:

- Default constructor.
- Constructor that takes the parameters:
    - A *string* filePath, an int nFields, an int nSamples
- Destructor.
- Const function getNumOfSamples() that takes no parameter and returns an int.
- Const function getNumOfFields() that takes no parameter and returns an int.
- Function emptySample() that takes no parameter and returns nothing.
- Const function printDatabase() that takes no parameter and returns nothing.
- Const function buildAssociation() that takes a *string* parameter called filePath. Nothing is returned.

Private:

- int value called numFields
- int value called numSamples
- A ***sample*** double pointer called samples

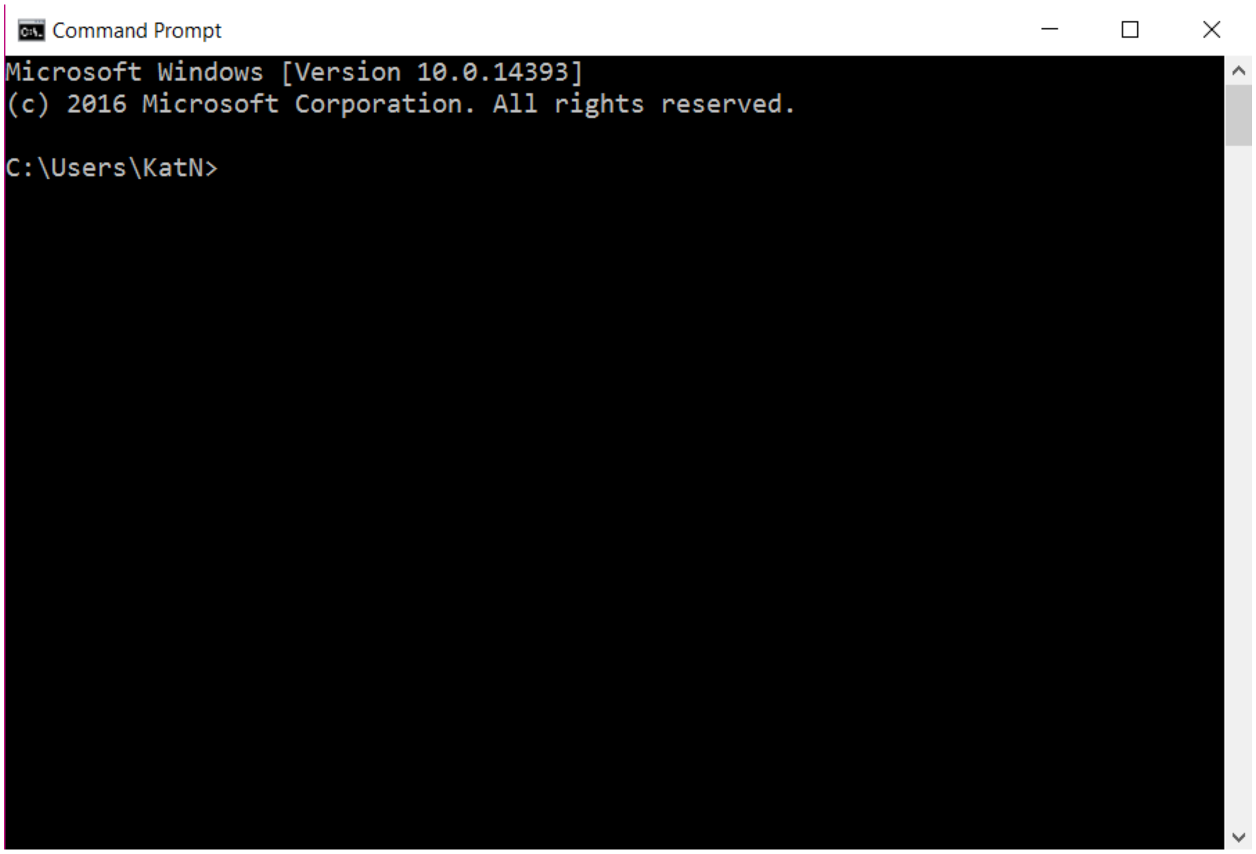
7. Save your work.
8. Coding Exercise:
   Try to implement database's member functions based on the definition below, then see the database.cpp file from Blackboard for an example solution.

   - Database() contains a pointer to an array of sample poiners. Sets array to 10 empty samples.
   - Database(*string* filePath, an int nFields, an int nSamples) opens the CSV of the path given and reads each row (sample) of the file. It should use the presence of commas in each line to separate data by fields. Each CSV will build a sample object. The document object stored in each sample will be initialized as well. All collected samples will be store in the database's samples double pointer.
   - ~database(): call emptySamples().
   - Const function getNumOfSamples(): returns number of rows.
   - Const function getNumOfFields(): returns number of fields.
   - Function emptySamples(): deletes each sample pointer in the samples** array.
   - Const function printDatabase() prints out each sample of the database.
   - Const function buildAssociation(): builds single text file called files.fit that stores a sample's filepath and class in this format:
     "pathToTextFile" + "\t" + classId

Using UDAT

With the association file generated, it's time to run your analysis on Udat! You first need to open a command line prompt. For Windows, a prompt window can be opened by either:

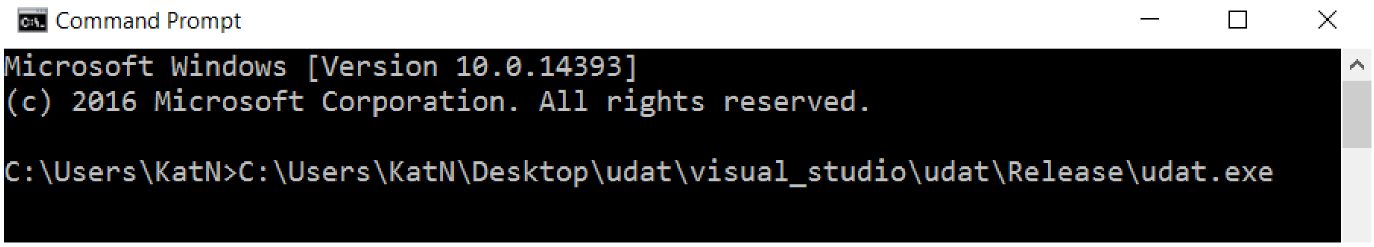
- Start → locate All Apps or All Programs → Windows System → Command Prompt
- Type "command prompt" or "cmd" into Window's search function found on the taskbar or in the Start menu → Command Prompt
- Win+R → Type "cmd" in the text box → Click OK



```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\KatN>
```

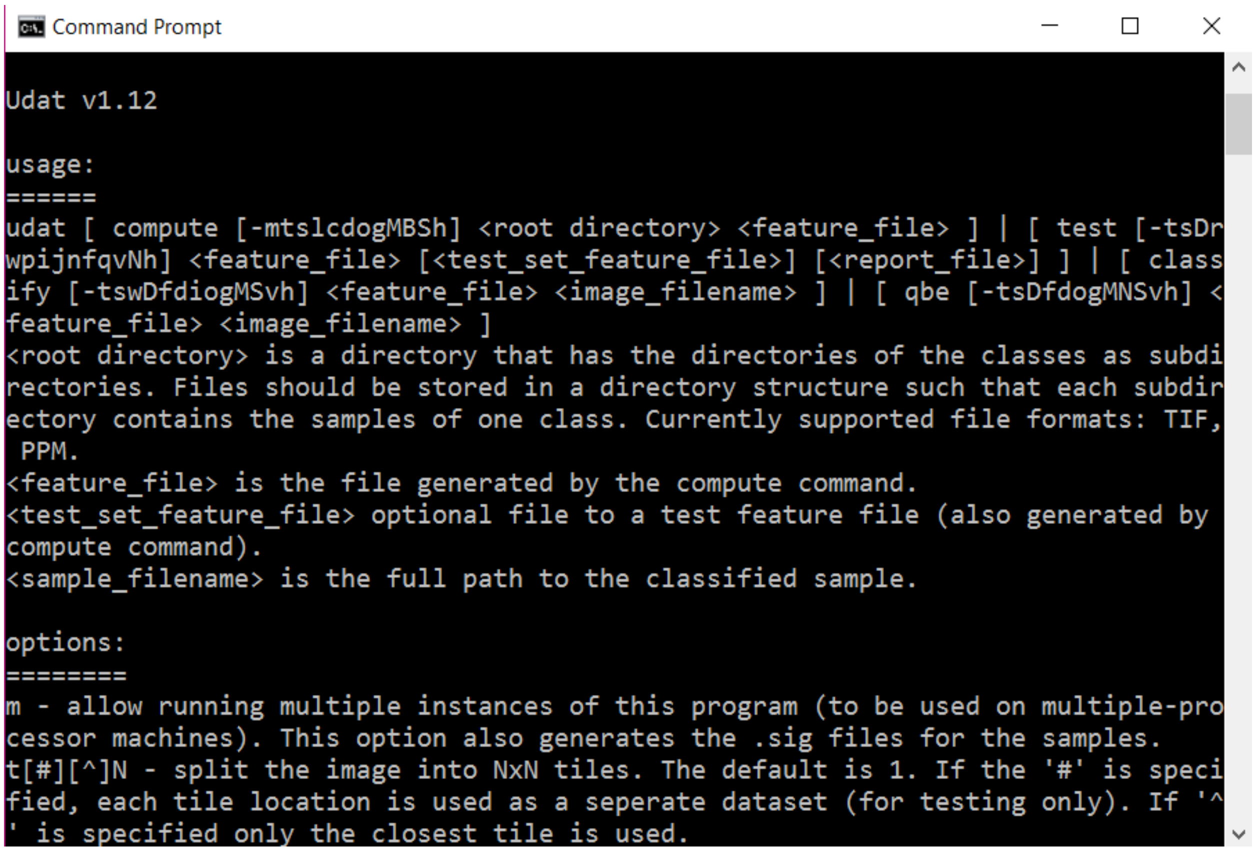
From the command line, start running Udat by typing in the path to your saved version of udat.exe. It should follow the format of:

C:\PATH TO UDAT FOLDER \udat\visual_studio\udat\Release\udat.exe



```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\KatN>C:\Users\KatN\Desktop\udat\visual_studio\udat\Release\udat.exe
```

Press Enter, and Udat's opening message should be displayed in the command prompt.



```
Udat v1.12

usage:
======
udat [ compute [-mtslcdogMBSh] <root directory> <feature_file> ] | [ test [-tsDr
wpijnfqvNh] <feature_file> [<test_set_feature_file>] [<report_file>] ] | [ class
ify [-tswDfdiogMSvh] <feature_file> <image_filename> ] | [ qbe [-tsDfdogMNSvh] <
feature_file> <image_filename> ]
<root directory> is a directory that has the directories of the classes as subdi
rectories. Files should be stored in a directory structure such that each subdir
ectory contains the samples of one class. Currently supported file formats: TIF,
 PPM.
<feature_file> is the file generated by the compute command.
<test_set_feature_file> optional file to a test feature file (also generated by
compute command).
<sample_filename> is the full path to the classified sample.

options:
========
m - allow running multiple instances of this program (to be used on multiple-pro
cessor machines). This option also generates the .sig files for the samples.
t[#][^]N - split the image into NxN tiles. The default is 1. If the '#' is speci
fied, each tile location is used as a seperate dataset (for testing only). If '^
' is specified only the closest tile is used.
```

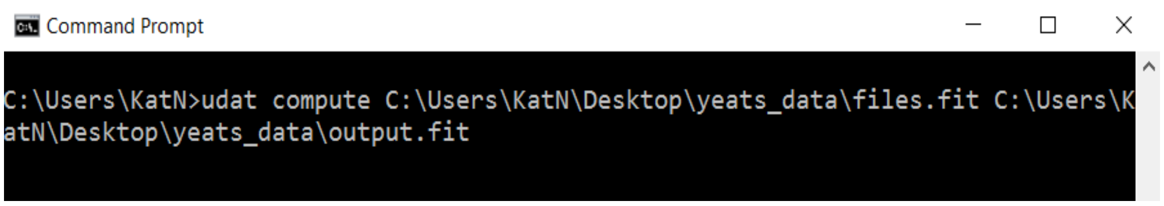
Read through the opening text to better understand the syntax used to run Udat, and what the software has to offer.

In general, you will not need to use all the syntax options made available. Below I will explain the rudimental options that you will use.

When running Udat, you will need to write only two command lines: udat compute and udat test. The compute line will process all the data from the training set, and extract the data's corresponding feature vectors. The test line will take the computed feature data as a comparison measure when attempting to classify each member of the test set.

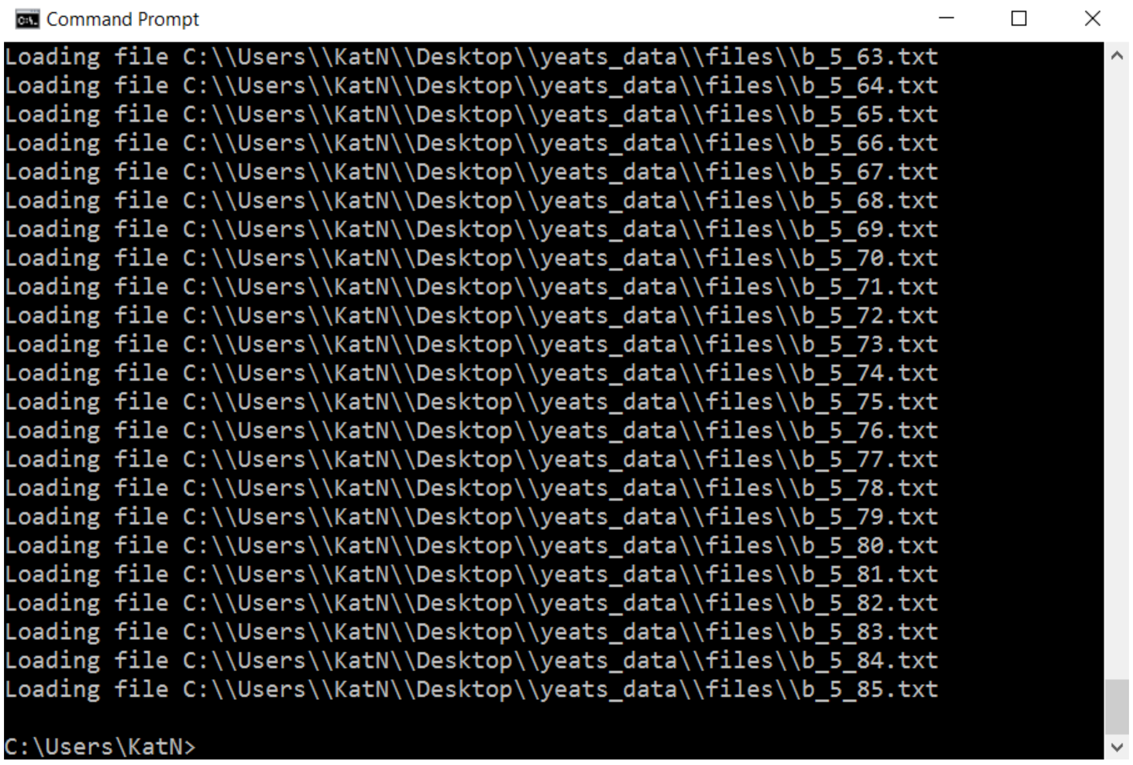
To run Udat on your document samples, you simply need these two lines

1. **udat compute C:\PATH TO INPUT ASSOCIATION FILE\files.fit C:\PATH FOR OUTPUT\output.fit** (Remember: the association file must be named files.fit for Udat to work.)



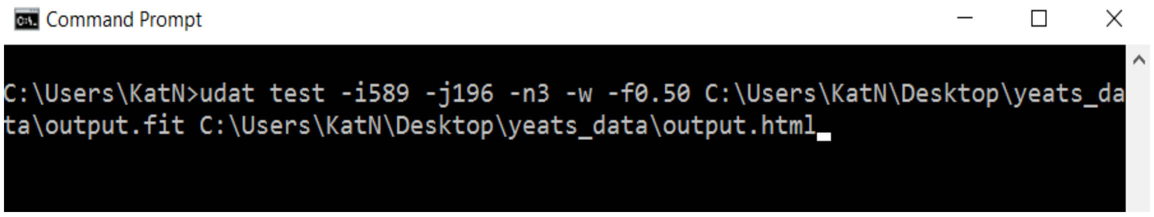
```
C:\Users\KatN>udat compute C:\Users\KatN\Desktop\yeats_data\files.fit C:\Users\K
atN\Desktop\yeats_data\output.fit
```

A successful execution of the compute command looks like this:


Command Prompt

```
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_63.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_64.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_65.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_66.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_67.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_68.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_69.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_70.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_71.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_72.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_73.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_74.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_75.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_76.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_77.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_78.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_79.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_80.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_81.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_82.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_83.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_84.txt
Loading file C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_5_85.txt

C:\Users\KatN>
```

2. **udat test –iVALUE –jVALUE -nVALUE -w -fVALUE C:\PATH TO OUTPUT FILE\output.fit C:\PATH FOR HTML OUTPUT\output.html**


Command Prompt

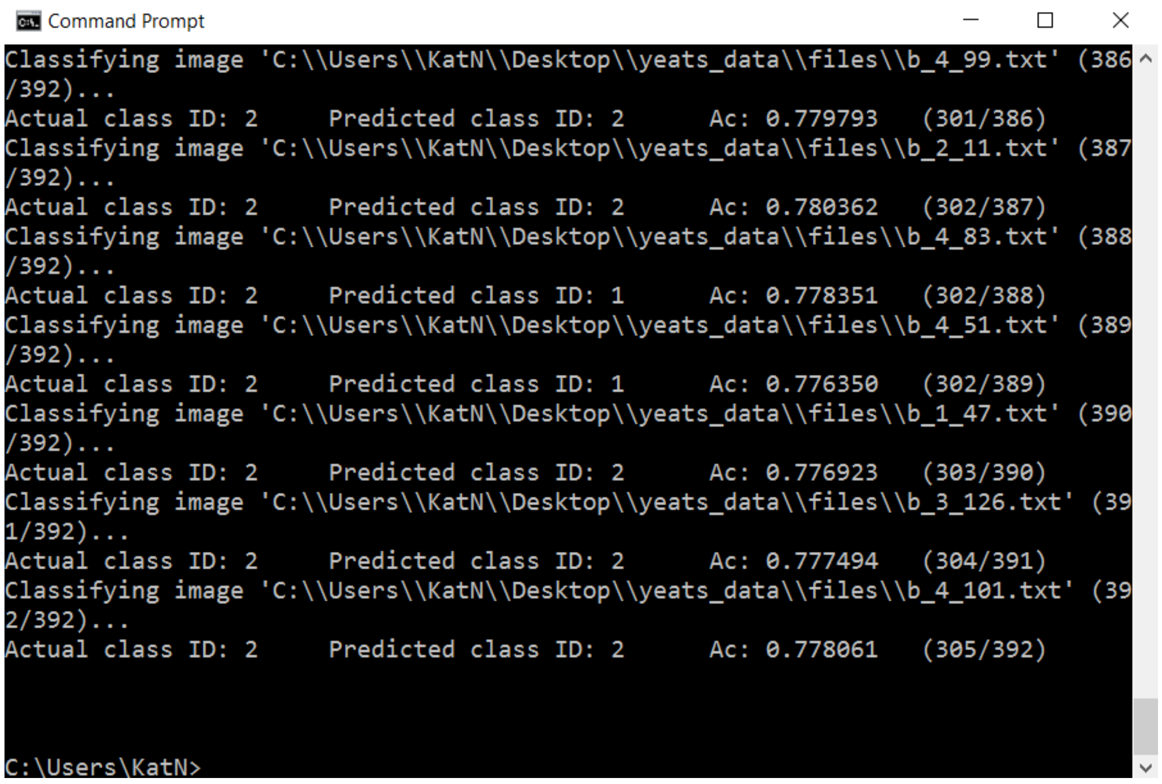
```
C:\Users\KatN>udat test -i589 -j196 -n3 -w -f0.50 C:\Users\KatN\Desktop\yeats_da
ta\output.fit C:\Users\KatN\Desktop\yeats_data\output.html
```

In writing the test line, there are 5 flags that you will use:
-i to define your training set size.
-j to define your test set size.
-n to define the number of times Udat is ran on your data. Each run uses a randomly selected training and test set.
-w to use WNN algorithm instead of WND
-f to define the percentage of available text features to test for.

The example above tests for 50% of the Udat's 220 features.

Alter the above flag options to suit your data and preferences. As a starting point for the values of i and j, set i as 75% of the number of files in found in your smallest class. Set j as 25% of your smallest class. Round appropriately.

A successful execution of the test command looks like this:

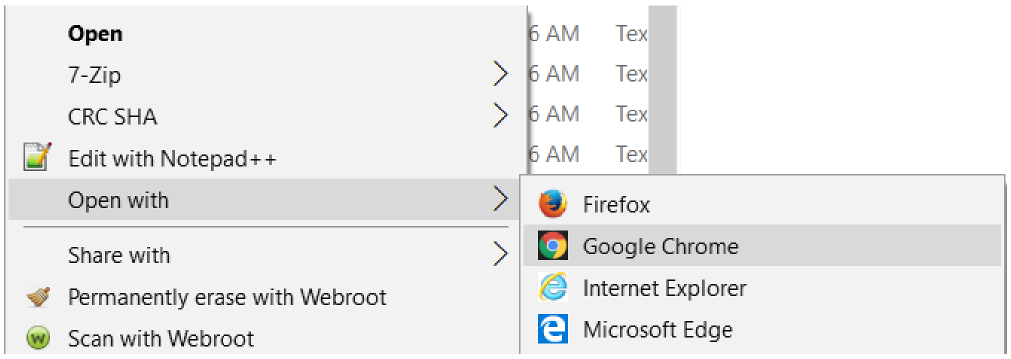


```
Classifying image 'C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_4_99.txt' (386
/392)...
Actual class ID: 2     Predicted class ID: 2      Ac: 0.779793    (301/386)
Classifying image 'C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_2_11.txt' (387
/392)...
Actual class ID: 2     Predicted class ID: 2      Ac: 0.780362    (302/387)
Classifying image 'C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_4_83.txt' (388
/392)...
Actual class ID: 2     Predicted class ID: 1      Ac: 0.778351    (302/388)
Classifying image 'C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_4_51.txt' (389
/392)...
Actual class ID: 2     Predicted class ID: 1      Ac: 0.776350    (302/389)
Classifying image 'C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_1_47.txt' (390
/392)...
Actual class ID: 2     Predicted class ID: 2      Ac: 0.776923    (303/390)
Classifying image 'C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_3_126.txt' (39
1/392)...
Actual class ID: 2     Predicted class ID: 2      Ac: 0.777494    (304/391)
Classifying image 'C:\\Users\\KatN\\Desktop\\yeats_data\\files\\b_4_101.txt' (39
2/392)...
Actual class ID: 2     Predicted class ID: 2      Ac: 0.778061    (305/392)



C:\Users\KatN>
```

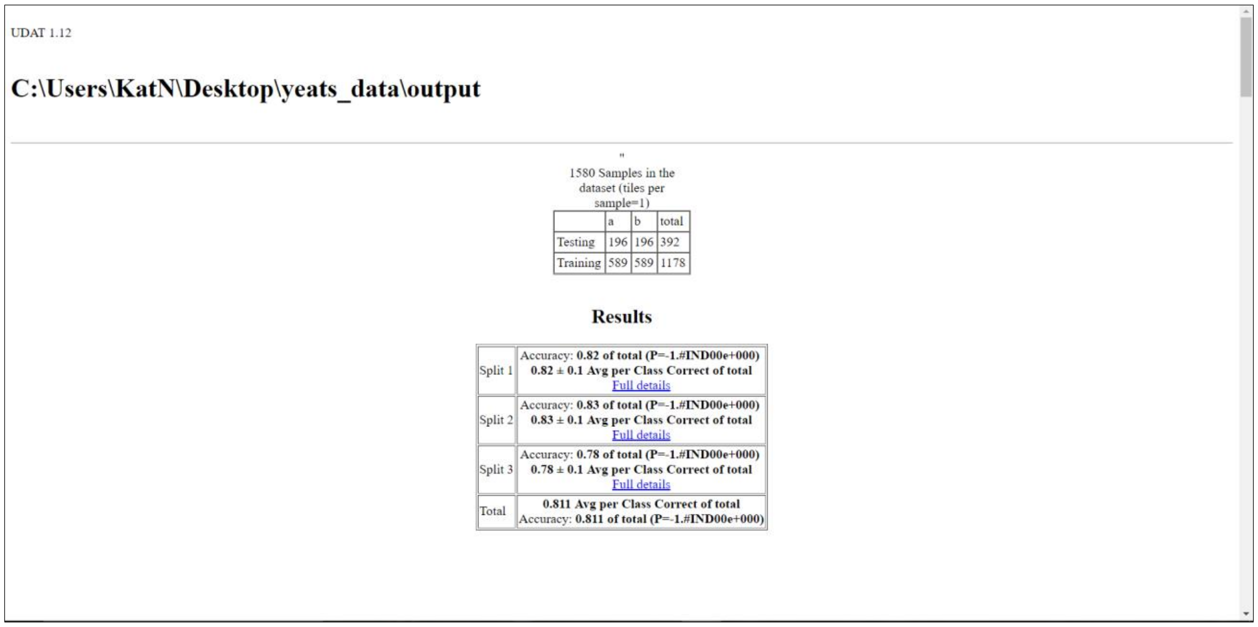
After running these two command lines, you should have an output.fit file and an output.html file stored in the paths you listed. Head over to the next section to learn how to analyze your output.html results.

Interpreting Udat Results

You have accomplished a complete run of Udat, but what did your text documents reveal? In the previous section, you had Udat generate a html file to store the run's results. Open the file by first locating the output.html file in file explorer, then right-clicking the file to choose Open with → your browser of choice.



When the results page opens up, it should look something like this:


UDAT 1.12

C:\Users\KatN\Desktop\yeats_data\output

"

1580 Samples in the dataset (tiles per sample=1)

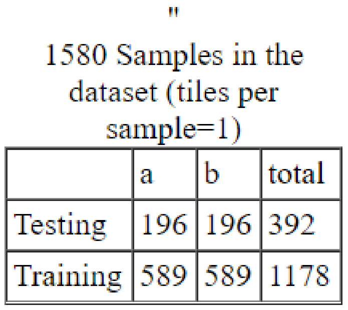
	a	b	total
Testing	196	196	392
Training	589	589	1178

Results

Split 1	Accuracy: **0.82 of total (P=-1.#IND00e+000)** **0.82 ± 0.1 Avg per Class Correct of total** Full details
Split 2	Accuracy: **0.83 of total (P=-1.#IND00e+000)** **0.83 ± 0.1 Avg per Class Correct of total** Full details
Split 3	Accuracy: **0.78 of total (P=-1.#IND00e+000)** **0.78 ± 0.1 Avg per Class Correct of total** Full details
Total	**0.811 Avg per Class Correct of total** Accuracy: **0.811 of total (P=-1.#IND00e+000)**

Below the file location of the output is a chart (Fig. 1) reiterating how many documents were used in the training set and test set from each class based on the values that were specified to Udat prior to running the program. I had 1,178 text files in the training set, and 392 text files in the test set.

Figure 1: Training and Test Set Split

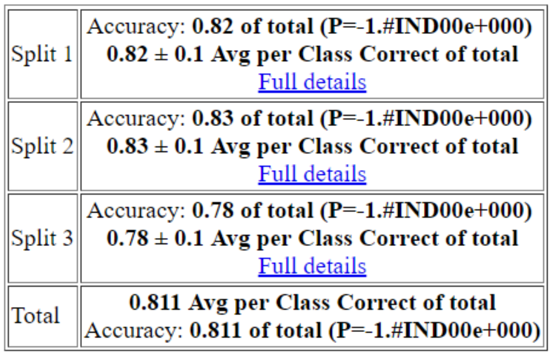

"

1580 Samples in the dataset (tiles per sample=1)

	a	b	total
Testing	196	196	392
Training	589	589	1178

Underneath the training/test set split is the official Udat results. Three important tables are listed in the mean results section: the accuracy table, the confusion matrix, and the average similarity matrix. In the accuracy table (Fig. 2), a decimal value represents how accurate Udat was in predicting the class attributes of the test set.

Figure 2: Udat's Accuracy Table Results



Split 1	Accuracy: **0.82 of total (P=-1.#IND00e+000)** **0.82 ± 0.1 Avg per Class Correct of total** Full details
Split 2	Accuracy: **0.83 of total (P=-1.#IND00e+000)** **0.83 ± 0.1 Avg per Class Correct of total** Full details
Split 3	Accuracy: **0.78 of total (P=-1.#IND00e+000)** **0.78 ± 0.1 Avg per Class Correct of total** Full details
Total	**0.811 Avg per Class Correct of total** Accuracy: **0.811 of total (P=-1.#IND00e+000)**

The classification accuracy can be summarized into a simple equation:

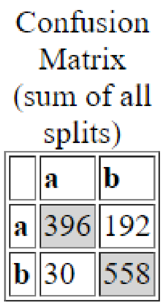
$$Accuracy = \frac{Number\ of\ Correctly\ Predicted\ Tweets}{Total\ Tweets\ Tested}$$

Next to the accuracy value is a ± sign introducing the **standard error** associated with the results. This standard error represents the variation between features means across the various Udat trial runs (in the case of my experiment, there were 3 runs). When Udat is run more than one different splits of the data, the accuracy results for each split is listed, along with an average accuracy value of all the trials.

Given the results seen in Fig. 2, Udat was able to accurately predict the class of a text sample an average of 81.1% of the time. This very good considering that the expected probability of correctly classifying a text between two classes is 50%.

The second table, the confusion matrix (Fig. 3), provides the number of times data from one class was labeled as data from another class during the classification process. The output combines (adds) the results from all split trials, if present. Rows headers in the confusion matrix represent the actual class and column headers represent the predicted class.

Figure 4: Udat's Confusion Matrix Results



Confusion Matrix (sum of all splits)

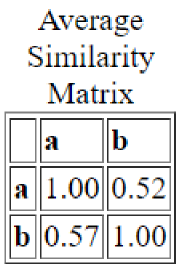
	a	**b**
a	396	192
b	30	558

Note: the average accuracy measurement from earlier can be calculated using the confusion matrix by adding the number of text files were correctly labeled (the diagonal values of the matrix), and dividing it by the total number of files tested (sum of all values in the matrix). To learn more about confusion matrices, head on over to this article: www.dataschool.io/simple-guide-to-confusion-matrix-terminology/

Analyzing my results, Yeat's work pre-1900s (*class a*) was correctly labeled 67.3% of the time, and his post-1910s work (*class b*) was correctly labeled 94.9% of the time. So pre-1900s documents were more often confused with post-1910s documents, then vice versa. What does this say about the two classes? It implies that there are strong distinguishing features found in *class b*, whereas *class a* takes a slightly more general form.

Lastly, the Similarity Matrix (Fig. 5), is a matrix that defines how similar classes are to one another through an algorithmic calculation of similarity using the feature vectors. A value of 1.00 means 100% similarity. Once again, row headers represent the actual class, and the column headers represent the class being tested for its similarity to the actual class.

Figure 5: Udat's Similarity Matrix Results



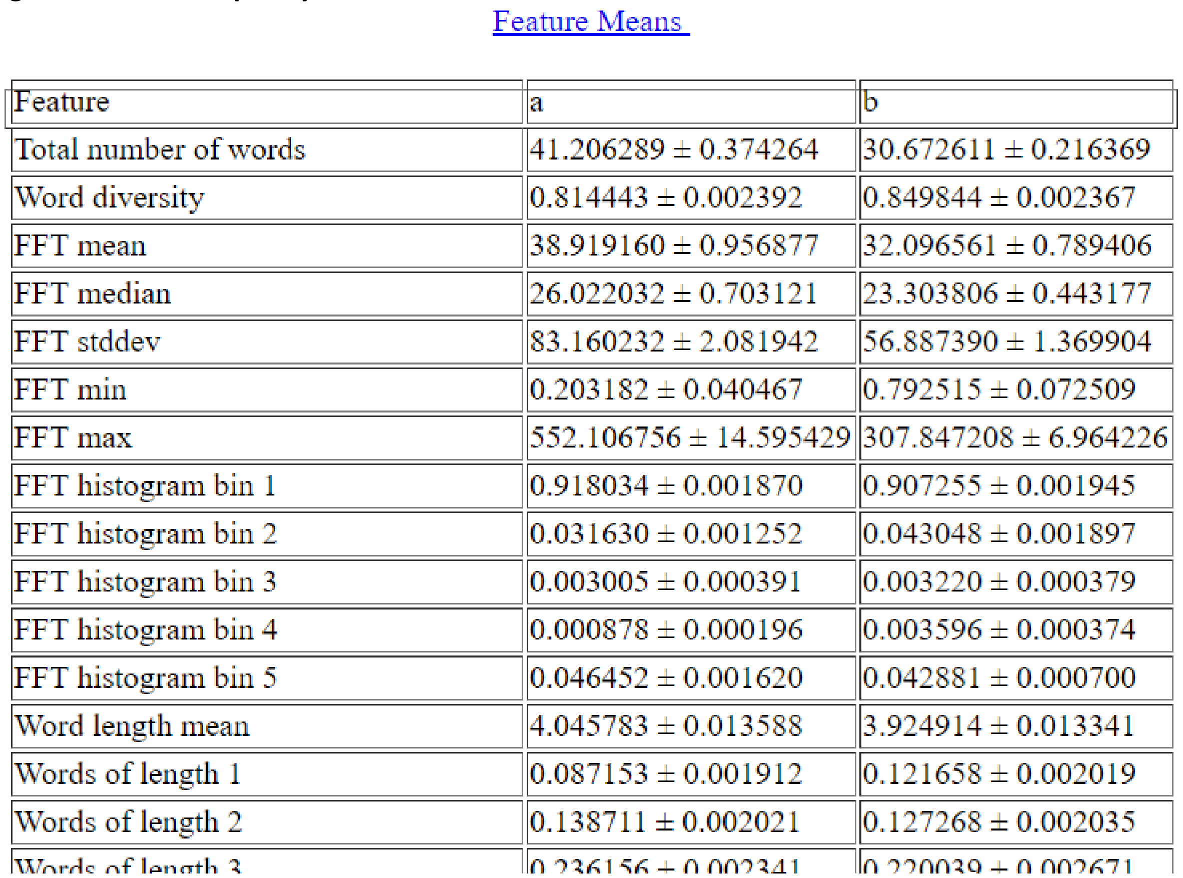
Average Similarity Matrix

	a	b
a	1.00	0.52
b	0.57	1.00

According to the results in Fig. 5, *class a* was 52% similar to *class b* and *class b* was 57% similar to *class a*. The small percentage difference between similarity values is to be expected when comparing only two classes.

After the three tables, is a link, titled *Feature Means*, which toggles the visibility of the feature means table (Fig. 6).

Figure 6: Udat's Frequency Means Table Results



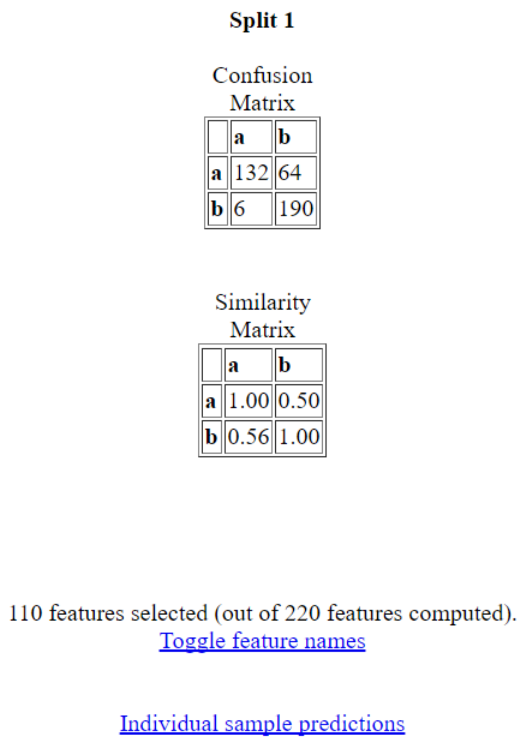
Feature Means

Feature	a	b
Total number of words	41.206289 ± 0.374264	30.672611 ± 0.216369
Word diversity	0.814443 ± 0.002392	0.849844 ± 0.002367
FFT mean	38.919160 ± 0.956877	32.096561 ± 0.789406
FFT median	26.022032 ± 0.703121	23.303806 ± 0.443177
FFT stddev	83.160232 ± 2.081942	56.887390 ± 1.369904
FFT min	0.203182 ± 0.040467	0.792515 ± 0.072509
FFT max	552.106756 ± 14.595429	307.847208 ± 6.964226
FFT histogram bin 1	0.918034 ± 0.001870	0.907255 ± 0.001945
FFT histogram bin 2	0.031630 ± 0.001252	0.043048 ± 0.001897
FFT histogram bin 3	0.003005 ± 0.000391	0.003220 ± 0.000379
FFT histogram bin 4	0.000878 ± 0.000196	0.003596 ± 0.000374
FFT histogram bin 5	0.046452 ± 0.001620	0.042881 ± 0.000700
Word length mean	4.045783 ± 0.013588	3.924914 ± 0.013341
Words of length 1	0.087153 ± 0.001912	0.121658 ± 0.002019
Words of length 2	0.138711 ± 0.002021	0.127268 ± 0.002035
Words of length 3	0.236156 ± 0.002341	0.220039 ± 0.002671

This charts records a feature's mean value for each class. Examples of a text features include the sentence length mean, frequency of music words, frequency of "!", etc. If more than one trial is run, the Feature Means table will show the average value of all the trials taken. Similar to the accuracy table, standard error is applied to the feature means. The length of this tables changes based on the value you set for the –f flag during the command line run of Udat. Using -f0.05, the Feature Means table of my results displays values for a total of 220 features.

The final section of output.html provides the individual data for each trial run. The individual results provided include a confusion matrix, a similarity matrix, a list of features names ranked by their significance as an identifier, and a table of individual sample predictions for each run (Fig. 7). This last table beneficial allows the user to compare the actual text to the predicted label, and try to discern what features of that text sample encouraged Udat to predict that particular class.

Figure 7: Results Provided for Each Trial Run



Split 1

Confusion Matrix

	a	b
a	132	64
b	6	190

Similarity Matrix

	a	b
a	1.00	0.50
b	0.56	1.00

110 features selected (out of 220 features computed).
Toggle feature names

Individual sample predictions

<u>My Results</u>
Based on the findings for my experiment, I wanted to test Udat's degree of accuracy. It is always a good idea to review if there was anything that could have skewed the results. In the case of my experiment, I was analyzing poems, plays, and stories written by Yeats. It could be possible that one class contained more of one type of literary work then the other. Hence, the text characteristics of that type of literary work could have skewed the results. For example, if one class had more stories in it, then files within that class could have longer sentences, more periods, etc.

To check the validity of Udat's accuracy levels, I removed any text files that were part of a play or story, and chose to analyze just poems. Running Udat, the level of accuracy made a sharp decline to 70.4%. Checking my data set again, I noticed that *class b* had around triple the pool of data to select from when creating training and test sets. The reason for the large decline in accuracy could be that *class b* now contained more collections of work than *class a*, so it had a more diverse set to find patterns in. So I reduced the number of collection in class b to one and made sure there was an equal number of files in *class b* and *class a*. Then, I ran Udat one more time. Accuracy increased to 82.4%. Overall, the accuracy of the first and last runs were very close.

Examining the feature means of my text, one of the most significant findings of my three trial runs, was that word diversity was constant fairly constant throughout. *Class a* had a word diversity value of roughly 0.81 and *class b* had a word diversity of roughly 0.85. To check its statistical significance, I used the t-test at http://www.graphpad.com/quickcalcs/ttest2/. This is great tool to check the statistical significance of any feature you would like analyze more. Inputting my data across the three Udat runs, the t-text indicated that the consistent difference in word diversity between *class a* and *class b* is extremely statistically significant with a P value of less than 0.0001. This indicated that the Yeat's breath of words choice, and possibly subject manner as well, increased in later years. So word diversity in his post-1910s work was greater than that of his pre-1900s work. To learn more about the t-test read http://blog.minitab.com/blog/adventures-in-statistics-2/understanding-t-tests-t-values-and-t-distributions.

What can you learn from your Udat results? Read through your oput.html page and see what your data is telling you. Try using GraphPad to see if you can find any statically significant features between your classes. If you need some starting points in your analysis, below are some basic questions to help you navigate your results. When you are done with your analysis, write down your findings and share them with the class! You have just completed your first text analysis project. When you are ready, move to the next section to see how you can easily enhance the scope of your project for even more insight on your topic of choice.

Quick Questions for Analyzing your Output:

- How accurate are Udat's predictions?
- How does the accuracy compare to the degree of accuracy that you would probably expect?
    - For example, if there are two classes, you would expect at least 50% accuracy.
    - Is Udat's accuracy too high or too low?
- Which classes are often confused with one another?
- Which classes are most similar to one another?
- What are the top 10 features ranked in each trail run?
- What are reoccurring high-rank features across the trial runs?
- What features values are statistically significant between the classes?

Conclusion

You've completed your first text analysis research project! Over the course of this project you've learned how to code classes, arrays, strings, pointers, and file input and output with C++. Gained experience collecting data, creating CSV files, and executing preprocessing. Finally, you ran the Udat text analysis software on your data and performed your own analysis of its results. You should feel proud of yourself. You've accomplished a lot of work, and I hope you had some fun exploring a topic that you have interest in as well.

Next Step

In terms of what is next, feel free to alter the program and CSV file that you created to use the text's author as the class value, instead of the year it was published. Then collect data from another source and compare the two different sources using Udat.

Appendix

A-1

Coding exercise from More Classes section:

Example Solution:

```
//main.cpp

#include <iostream>
#include “document.h”
using namespace std;

int main()
{
   document doc1;
   doc1.setText(“ Enter your text ”);
   cout << doc1.getText() << endl;

   document doc2( “Enter new text” );
   document * doc3 = new document();

   return 0;
};
```

B-1

Alternative approach to initializing the array:

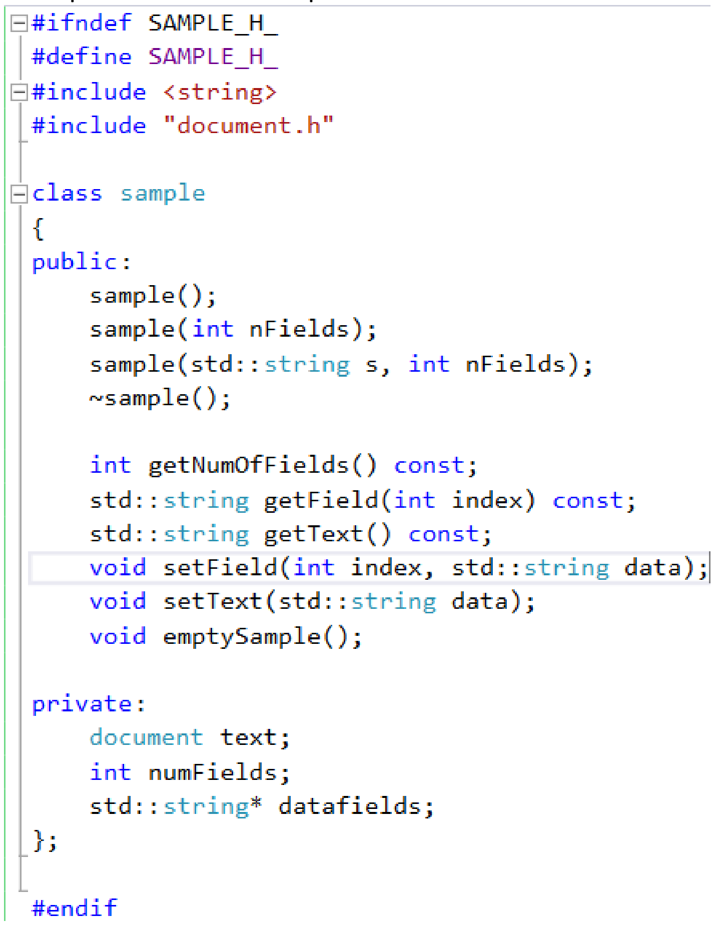
Example Solution:

```
//main.cpp
```

```
#include <iostream>
#include “document.h”
using namespace std;

int main()
{
   document doc1;
   doc1.setText(“ Enter your text ”);
   cout << doc1.getText() << endl;

   document doc2( “Enter new text” );
   document * doc3 = new document();

   document collection[5] = {doc1, doc2};

   cout << collection[1].getText() << endl;

   return 0;
};
```

B-2

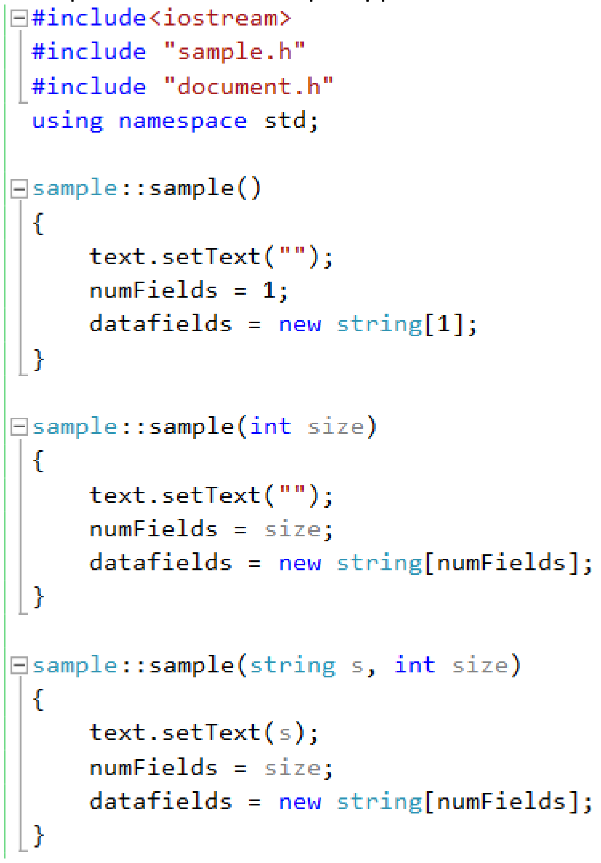
Example Solution for sample.h file:



```
#ifndef SAMPLE_H_
#define SAMPLE_H_
#include <string>
#include "document.h"

class sample
{
public:
    sample();
    sample(int nFields);
    sample(std::string s, int nFields);
    ~sample();

    int getNumOfFields() const;
    std::string getField(int index) const;
    std::string getText() const;
    void setField(int index, std::string data);
    void setText(std::string data);
    void emptySample();

private:
    document text;
    int numFields;
    std::string* datafields;
};

#endif
```

B-3

Example Solution for sample.cpp file:

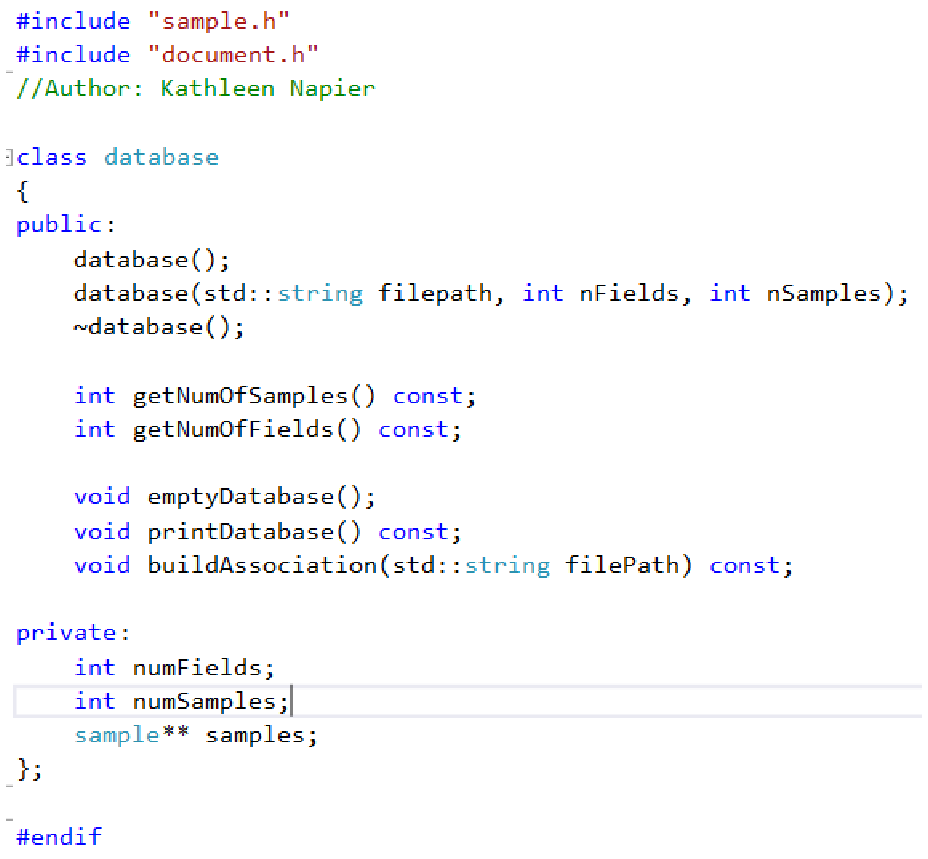


```
#include<iostream>
#include "sample.h"
#include "document.h"
using namespace std;

sample::sample()
{
    text.setText("");
    numFields = 1;
    datafields = new string[1];
}

sample::sample(int size)
{
    text.setText("");
    numFields = size;
    datafields = new string[numFields];
}

sample::sample(string s, int size)
{
    text.setText(s);
    numFields = size;
    datafields = new string[numFields];
}
```

```
sample::~sample()
{
    emptySample();
}

int sample::getNumOfFields() const
{
    return numFields;
}

string sample::getField(int index) const
{
    if (index <= numFields && index >= 0)
    {
        return datafields[index];
    }
    else
    {
        cout << "Error: Out-of-bound index when retrieving data." << endl;
        return "NULL";
    }
}

void sample::setField(int index, string data)
{
    if (index <= numFields && index >= 0)
    {
        datafields[index] = data;
    }
    else
    {
        cout << "Error: Out-of-bound index when setting data." << endl;
    }
}

string sample::getText() const
{
    return text.getText();
}

void sample::setText(string data)
{
    text.setText(data);
}

void sample::emptySample()
{
    delete[] datafields;
}
```

C-1

Example solution of database.h:



```cpp
#include "sample.h"
#include "document.h"
//Author: Kathleen Napier

class database
{
public:
    database();
    database(std::string filepath, int nFields, int nSamples);
    ~database();

    int getNumOfSamples() const;
    int getNumOfFields() const;

    void emptyDatabase();
    void printDatabase() const;
    void buildAssociation(std::string filePath) const;

private:
    int numFields;
    int numSamples;
    sample** samples;
};

#endif
```