**WEEK 9**

### ****INSTRUCTIONS****

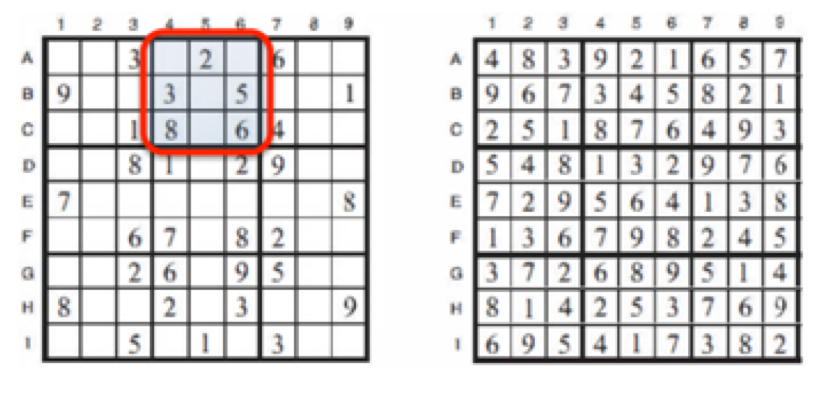
In this assignment you will focus on constraint satisfaction problems. You will be implementing the AC-3 and backtracking algorithms to solve Sudoku puzzles. The objective of the game is just to ﬁll a 9 x 9 grid with numerical digits so that each column, each row, and each of the nine 3 x 3 sub-grids (also called boxes) contains one of all of the digits 1 through 9. If you have not played the game before, you may visit [**sudoku.com**](http://www.sudoku.com/) to get a sense of how the game works.

Please read all sections of the instructions carefully. In particular, note that you have a total of **5 submission** attempts.

**I.** Introduction  
**II.** What You Need To Submit  
**III.** AC-3 Algorithm  
**IV.** Backtracking Algorithm  
**V.** Important Information  
**VI.** Before You Submit

### ****I. Introduction****

Consider the Sudoku puzzle as pictured below. There are 81 **variables** in total, i.e. the tiles to be filled with digits. Each variable is named by its **row** and its **column**, and must be assigned a **value** from 1 to 9, subject to the constraint that no two cells in the same row, column, or box may contain the same value.



In designing your classes, you may find it helpful to represent a Sudoku board with a Python dictionary. The keys of the dictionary will be the variable names, each of which corresponds directly to a location on the board. In other words, we use the variable names **Al** through **A9** for the top row (left to right), down to **I1** through **I9** for the bottom row. For example, in the example board above, we would have sudoku["**B1**"] = **9**, and sudoku["**E9**"] = **8**. This is the highly suggested representation, since it is easiest to frame the problem in terms of **variables**, **domains**, and **constraints** if you start this way. However, you can choose other data structures if you prefer.

### ****II. What You Need To Submit****

Your job in this assignment is to write driver.py, which intelligently solves Sudoku puzzles. Your program will be executed as follows:

$ python driver.py <input\_string>

In the starter code folder, you will find the file sudokus\_start.txt, containing hundreds of sample Sudoku puzzles to be solved. Each Sudoku puzzle is represented as a single line of text, which starts from the top-left corner of the board, and enumerates the digits in each tile, row by row. In this assignment, we will use the number **zero** to indicate tiles that have not yet been filled. For example, the Sudoku board in the diagram shown above is represented as the string:

00302060090030005001001806400... (and so on)

When executed as above, replacing "<input\_string>" with any valid string representation of a Sudoku board (for instance, taking any Sudoku board from sudokus\_start.txt), your program will generate a file called output.txt, containing **a single line** of text representing the finished Sudoku board and the algorithm name (**AC3** or **BTS**, explained later) which solved the Sudoku board. You must use **a single white space** as a delimiter between the board and the algorithm name. For example, output.txt looks like:

167523849984176523325489671798315264642798135531642798476831952213957486859264317 BTS

(single line, separated by a single white space)

Since this board is solved, the string representation will contain no zeros. You may test your program extensively by using sudokus\_finish.txt, which contains the solved versions of all of the same puzzles.

**Note on Python 3**

As usual, if you choose to use Python 3, then name your program driver\_3.py. In that case, the grader will automatically run your program using the python3 binary instead. Please only submit one version. **If you submit both versions, the grader will only grade one of them, which probably not what you would want.** To test your algorithm in Python 3, execute the game manager as follows:

$ python3 driver\_3.py <input\_string>

### ****III. AC-3 Algorithm (AC3)****

First, implement the **AC-3 algorithm**. Test your code on the provided set of puzzles in sudokus\_start.txt. To make things easier, you can write a separate wrapper script (bash, or python) to loop through all the puzzles to see if your program can solve them. As shown in sudokus\_finish.txt, there are only 2/400 Sudoku boards which can be solved AC3 alone. Is this expected or unexpected?

### ****IV. Backtracking Algorithm (BTS)****

Now, implement **backtracking** using the **minimum remaining value** heuristic. The order of values to be attempted for each variable is up to you. When a variable is assigned, apply **forward checking** to reduce variables domains. Test your code on the provided set of puzzles in sudokus\_start.txt. Can you solve all puzzles now?

### ****V. Important Information****

Please read the following information carefully. Before you post a clarifying question on the discussion board, make sure that your question is not already answered in the following sections.

**1. Precedence over BTS**

**To check how powerful BTS is compared to AC3, you must execute AC-3 algorithm before Backtracking Search algorithm. That is, your program looks like this:**

assignment = AC3(given\_sudoku\_board)  
if (solved(assignment))  
          return "<filled sudoku board>" + " AC3"  
assignment = BTS(given\_sudoku\_board)  
          return "<filled sudoku board>" + " BTS"

**2. Test-Run Your Code**

To avoid wasting submission attempts, please test-run your code on Vocareum, and make sure it successfully produces an output file with the correct format. You can do this by hitting the **RUN** button, which simply executes your program with a sample input string containing a valid starting Sudoku board. After you hit **RUN**, when your program terminates, you should locate the output file within your working directory. Make sure the board and the algorithm name is separated by a single white space.

**3. Grading Submissions**

We will test your final program on **20 test cases**. You can assume all test cases can be solved at least by BTS. Some of test cases might be solved by AC3 alone. Each input test case will be rated **5 points** for a successfully solved board, and zero for any other resultant output. In sum, your submission will be assessed out of a total of 100 points. The test cases are no different in nature than the hundreds of test cases already provided in your starter code folder, for which the solutions are also available. If you can solve all of those, your program will most likely get full credit.

**4. Time Limit**

By now, we expect that you have a good sense of appropriate data structures and object representations. Naive brute-force approaches to solving Sudoku puzzles may take minutes, or even hours, to [possibly never] terminate. However, a correctly implemented backtracking approach as specified above should take **well under a minute** per puzzle. The grader will provide some breathing room, but programs with much longer running times will be killed.

### ****USE OF VOCAREUM****

This assignment uses Vocareum for submission and grading. Vocareum comes equipped with an editing environment that you may use to do your development work. You are **NOT** required to use the editor. In particular, you are free to choose your favorite editor / IDE to do your development work on. When you are done with your work, you can simply upload your files onto Vocareum for submission and grading.

However, your assignments will be graded on the platform, so you **MUST** make sure that your code executes without error on the platform. In particular, do not use any additional third-party libraries and packages. We do not guarantee that they will work on the platform, even if they work on your personal computer. For the purposes of this project, everything that comes with the standard Python library should be more than sufficient.

**WEEK 11**

**INSTRUCTIONS**

Congratulation on making it to the last programming project. By coming this far, we assume that you have accumulated formidable knowledge in both traditional Artificial Intelligence (AI) and modern Machine Learning (ML), and from now on we will treat you as such. This assignment intends to give you a flavor of a real world AI/ML application, which often require to gather the raw data, do preprocessing, design suitable ML algorithms and implement the solution. Today, we touch on an active research area in Natural Language Processing (NLP), sentiment analysis.

Given the exponentially growing of online review data (Amazon, IMDB and etc), sentiment analysis becomes increasingly important. We are going to build a sentiment classifier, i.e., evaluating a piece of text being either positive or negative.

The "Large Movie Review Dataset"(\*) shall be used for this project. The dataset is compiled from a collection of 50,000 reviews from IMDB on the condition there are no more than 30 reviews each movie. Number of positive and negative reviews are equal. Negative reviews have scores lesser or equal 4 out of 10 while a positive review greater or equal 7 out of 10. Neutral reviews are not included on the other hand. Then, 50,000 reviews are divided evenly into the training and test set.

*\*Dataset is credited to Prof. Andrew Mass in the paper, Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011).*[*Learning Word Vectors for Sentiment Analysis.*](http://ai.stanford.edu/~amaas/papers/wvSent_acl2011.pdf)*The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011).*

**I. Instruction**

Up until now, most of the course projects have been requiring you to implement algorithms discussed in lectures. This assignment is going to introduce a few advanced concepts of which implementations demand a non-trivial programming expertise. As such, before reinventing the wheel, we would advise you to first explore the incredibly powerful existing Python libraries. The following two are highly recommended:

* <http://scikit-learn.org/stable/>
* <http://pandas.pydata.org/>

**Stochastic Gradient Descent Classifier**

In this project, we will train a Stochastic Gradient Descent Classifier. Recalled from the Machine Learning project, you were asked to implement a gradient descend update algorithm for linear regression. While gradient descend is powerful, it can be prohibitively expensive when the dataset is extremely large because every single data point  needs to be processed.

However, it turns out when the data is large, rather than the entire dataset, SGD algorithm performs just as good with a small random subset of the original data. This is the central idea of Stochastic SGD and particarly handy for the text data since corpus are often humongous.

You should read sklearn document and learn how to use a SGD classifier. For adventurers, you are welcome to manually implement SGD yourself. Wikipedia provides a good first reference, <https://en.wikipedia.org/wiki/Stochastic_gradient_descent>.

***Data Preprocessing***

The training data is provided in the directory "../resource/lib/publicdata/aclImdb/train/" of Vocareum. If you wish to download the data to your local machine for inspections, use the following link: <http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz>.

Your first task is explore this directory. There are two sub-directories **pos/** for positive texts and **neg/** for negative ones. **You do not need to worry about unsup/, and you do not ned them.**

Now **combine the raw database into a single csv files,** “**imdb\_tr.csv**”. The csv file should have three columns, **"row\_number"** and **“text” and “polarity”**. The column **“text”** contains review texts from the aclImdb database and the column **“polarity”** consists of sentiment labels, 1 for positive and 0 for negative. An example of "imdb.tr.csv" is provided in the workspace.

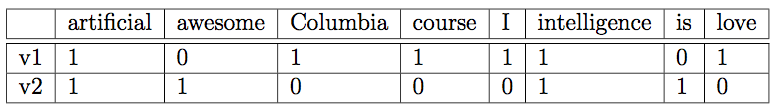
In addition, common English stopwords should be removed**.**An English stopwords reference are provided in your Vocareum work space for your reference. Your driver.py (as explained below) **will have access to it during run time**.

***Unigram Data Representation***

The very first step in solving any NLP problem is finding a way to represent the text data so that machines can understand. A common approach is using a document-term vector where each document is encoded as a discrete vector that counts occurrences of each word in the vocabulary it contains. For example, consider two one-sentence documents:

* d1: “I love Columbia Artificial Intelligence course.
* d2: “Artificial Intelligence is awesome”

The vocabulary V = {artificial, awesome, Columbia, course, I, intelligence, is, love} and two documents can be encoded as v1 and v2 as follow:



Hint: When building our model you should assume no access to the test data. Then what if there are words that appear only in test data but not in training data? The features will mismatch if you include those. Therefore, when extracting features in the test set, you should only **use the vocabulary that was used in the training set.**

If you wish to know more, start from here [https://en.wikipedia.org/wiki/Document-term\_ matrix](https://en.wikipedia.org/wiki/Document-term_%20matrix). This data representation is also called **a unigram model**.

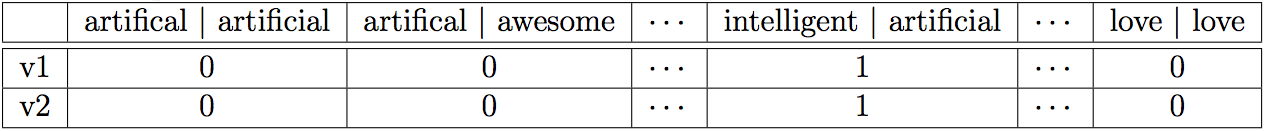
Now, write a python function to transform text column in **imdb\_tr.csv** into a term-document matrices using uni- gram model then train a **Stochastic Gradient Descent (SGD) classifier** whose loss=“hinge” and penalty=“l1” on this data.

On the other hand, in the driver.py, you will also find the link to ***"../resource/lib/publicdata/imdb\_te.csv"*** which is our benchmark file for the performance of the trained classifier. "***imdb\_te.csv"***has two columns: **"row\_number"** and **"text".** The column **"polarity"** is excluded and your job is to use the trained SGD classifier to predict this information. You should transform **imdb\_te.csv** using unigram data model as well and use the trained SGD to predict the converted test set. Predictions must be formatted line by line and stored in "**unigram.output.txt"**in your Vocareum workspace. An example of the output file is provided for your benefits.

**If you wish to run the test in your local machine, download the following** [**test file**](https://prod-edxapp.edx-cdn.org/assets/courseware/v1/9dbe589c9a231b5174729e059a17e8eb/asset-v1:ColumbiaX+CSMM.101x+1T2018+type@asset+block/imdb_te.csv.zip)**.**

***Bigram Representation***

A more sophisticated data representation model is the bigram model where occurrences depend on a sequence of two words rather than an individual one. Taking the same example like before, v1 and v2 are now encoded as follow:



Instead of enumerating every individual words, bigram counts the number of instance a word following after another one. In both d1 and d2 “intelligence” follows “artificial” so v1(intelligence | artificial) = v2(intelligence | artificial) = 1. In contrast, “artificial” does not follow “awesome” so v1(artificial | awesome) = v2(artificial | awesome) = 0.   
Repeat the same exercise from Unigram for the Bigram Model Data Representation and produce the test prediction file "**bigram.output.txt" .**

***Tf-idf:***

Sometimes, a very high word counting may not be meaningful. For example, a common word like “say” may appear 10 times more frequent than a less-common word such as “machine” but it does not mean “say” is 10 times more relevant to our sentiment classifier. To alleviate this issue, we can instead use **term** **frequency tf[t] = 1 + log(f[t,d] )** where **f[t,d]** is the count of term t in document d. The log function dampens the unwanted influence of common English words.

Inverse document frequency (idf) is a similar concept. To take an example, it is likely that all of our training documents belong to a same category which has specific jargons. For example, Computer Science documents often have words such as computers, CPU, programming and etc  appearing over and over. While they are not common English words, because of the document domain, their occurrences are very high. To rectify, we can adjust using **inverse term frequency *idf[t] = log( N / df[t] )***where **df[t]** is the number of documents containing the term t and N is the total number of document in the dataset.

Therefore, instead of just word frequency, tf-idf for each term t can be used, **tf-idf[t] = tf[t] ∗idf[t].**

Repeat the same exercise as in the Unigram and Bigram data model but apply tf-idf this time to produce test prediction files, "**unigramtfidf.output.txt"**and "**bigramtfidf.output.txt"**

**II. What you need to submit:**

Your task in this assignment is to write driver.py to produce sentiment predictions over the **imdb\_te.csv** by various text data representation (unigram, unigram with tf-idf, bigram and bigram with tf-idf). Please ensure your driver.py write the predictions to the following files **during the run time** (one-time outputs are not accepted):

* unigram.output.txt
* unigramtfidf.output.txt
* bigram.output.txt
* bigramtfidf.output.txt

Be very ***precise*** with these file names because the auto-grader will rerun your driver.py and look for them for evaluation. As usual, your program will be run as follows:

$python driver.py

If you want to use Python 3 then simply rename driver.py to driver\_3.py and your program will be executed as:

$python3 driver\_3.py

It is highly recommended that before submission you should perform some sanity check so you will not waste your time and opportunity to submit. Below are something you want to keep in mind:

- The name of your program file correspond with the expected, exactly

- The name of the output file generated by your program

- The libraries that you are using in your program be allowed (only standards lib)

- The way you read the training and testing data is correct (Be aware of **headers**! Do not get off-by-one error!)

- You have performed cross validation on your model

Note:  Our grade will **not**call imdb\_data\_preprocess() ourselves. You will need to do data processing under *if \_\_name\_\_ == "\_\_main\_\_":*by yourself in the driver.