Homework 1

10-405/605: Machine Learning with Large Datasets

Due Monday, January 27th at 11:59PM Eastern Time

Submit your solutions via Gradescope, **with your solution to each subproblem on a separate page**, i.e., following the template below.

IMPORTANT: Be sure to highlight where your solutions are for each question when submitting to Gradescope otherwise you will be marked 0 and will need to submit regrade request for every solution un-highlighted in order for fix it!

Note that Homework 1 consists of two parts: this written assignment, and a programming assignment. Remember to fill out the collaboration section found at the end of this homework as per the course policy.

All students are required to complete the following:

1. Written Section
   (a) Effective Sparse Indexing *[30 points]*
   (b) Combiners and the "last reducer" problem *[20 points]*
2. Programming Section
   (a) Entity Resolution *[20 points]*
   (b) Streaming Naive Bayes *[10 points]*
   (c) Naive Bayes on Spark *[20 points]*

All students are required to complete **all** sections of the homework. Your homework score will be calculated as a percentage over the maximum points you can earn, which is 100.

1 Written Section [50 Points]

1.1 Effective Sparse Indexing [30 Points]

1.1.1 Motivation

The programming part of this assignment uses Spark to find similar pairs of documents, and part 4 discusses how to use *inverted indices* to make this more efficient.

In this part, we will look at ways inverted indices can be used to efficiently find candidate *retrieval documents* that might be highly similar to a given *query document.*

1.1.2 Basic Notation

We will model words/tokens (sometimes called terms) as integers t, where $1 \leq t \leq V$, so V is the *vocabulary size*. We will model documents as vectors of length V, where component t of that vector is the weight of term t in the document it encodes. We will use the following notation:

$$\begin{array}{lll} \boldsymbol{q} = & \langle q_1, \ldots, q_t, \ldots, q_V \rangle & \text{a query document} \\ \boldsymbol{r} = & \langle r_1, \ldots, r_t, \ldots, r_V \rangle & \text{a retrieved document} \\ R = & \{r^{(1)}, \ldots, r^{(d)}, \ldots, r^{(N)}\} & \text{a corpus} \end{array}$$

We assume that all term weights are non-negative and that the weight for term t is zero if and only if term t does not appear in document d. This means that the vectors above will be sparse (i.e., mostly zeros). We call the set of non-zero components of $\boldsymbol{r}$ the *support* of $\boldsymbol{r}$.

$$\text{support}(\boldsymbol{r}) \equiv \{t : t \text{ appears in doc } \boldsymbol{r}\} \equiv \{t : r_t \neq 0\}$$

The *inverted index* for a term t is the set of documents that contain t:

$$R_t \equiv \{d : t \in \text{support}(\boldsymbol{r}^{(d)})\}$$

Note that if you think of R as a matrix, where the documents are rows, and let $R[:, t]$ be the t-th column of R, then R_t is just the indices of $R[:, t]$ with non-zero entries.

An *index set* is simply any subset of terms: i.e., S is an index set iff $S \subseteq \{1, \ldots, V\}$. We can define the "inverted index" for an index set S as the set of documents in R that contain any term in S:

$$R_S \;\equiv\; \{d : S \cap \text{support}(\boldsymbol{r}^{(d)}) \neq \emptyset\} \;\equiv\; \bigcup_{t \in S} R_t$$

It's easy to design data structures where you can efficiently find R_t given t, and if you do that, it's easy to find the documents in R_S using $R_S = \cup_{t \in S} R_t$.

1.1.3 Other convenient concepts

An upper-bound vector. We will make use of a vector $\boldsymbol{u}$ which contains the maximum weight of each term over all documents in the corpus R:

$$\begin{array}{rll} & \boldsymbol{u} = & \langle u_1, \ldots, u_t, \ldots, u_V \rangle \\ \text{where} & u_t \equiv & \max_{\boldsymbol{r} \in R} r_t \end{array}$$

Vectors restricted to an index set. An index set S can also be thought of as a vector $\boldsymbol{s}$ where

$$s_t = \begin{cases} 1 & \text{if } t \in S \\ 0 & \text{else} \end{cases}$$

Let $\odot$ denote the Hademard (aka component-wise) product, i.e., $\boldsymbol{r} \odot \mathrm{s} \equiv \langle \mathrm{r}_1\mathrm{s}_1, \ldots, \mathrm{r}_\mathrm{t}\mathrm{s}_\mathrm{t}, \ldots, \mathrm{r}_\mathrm{V}\mathrm{s}_\mathrm{V} \rangle$. It is useful to talk about vectors of the form $\boldsymbol{r} \odot \boldsymbol{s}$. Below we denote these vectors as r_S, where S is the index set encoded by $\boldsymbol{s}$, i.e.,

$$\boldsymbol{r}_S \equiv \boldsymbol{r} \odot \boldsymbol{s} = \langle r_1 s_1, \ldots, r_t s_t, \ldots, r_V s_V \rangle$$

Intuitively, $\boldsymbol{r}_S$ is simply $\boldsymbol{r}$ with all the terms not in S set to have weight zero.

Linearity of inner products. For many weighting schemes, the inner product between vectors is a useful measure of document similarity. Recall that for any scalars a, b and vectors $\boldsymbol{v}$, $\boldsymbol{w}$, $\boldsymbol{x}$,

$$(a\boldsymbol{v} + b\boldsymbol{w}) \cdot \boldsymbol{x} = a(\boldsymbol{v} \cdot \boldsymbol{x}) + b(\boldsymbol{w} \cdot \boldsymbol{x})$$

1.1.4 Problems

(a) *[5 points]* Let's first show that all documents that have non-zero similarity to $\boldsymbol{q}$ are in the inverted index of some term t that appears in $\boldsymbol{q}$.

Let $S = \text{support}(\boldsymbol{q})$. Show that if $d \notin R_S$, then $\boldsymbol{q} \cdot \boldsymbol{r}^{(d)} = 0$.

(b) *[4 points]* Consider the following more general statement.

Let S be a subset of the support terms in $\boldsymbol{q}$, i.e., $S \subseteq \text{support}(\boldsymbol{q})$. Show that if $d \notin R_S$, then $\boldsymbol{q}_S \cdot \boldsymbol{r}^{(d)} = 0$.

(c) *[4 points]* Suppose $S' \subset \text{support}(\boldsymbol{q})$ is some set of index terms that does *not* include everything in the support of $\boldsymbol{q}$, and let $t \in (\text{support}(\boldsymbol{q}) - S')$ be some term in the query document that is not in S'. Assume that every term in the query $\boldsymbol{q}$ does appear in the corpus at least once (with non-zero weight).

Yes or No: must it be true that there exists a document $d \notin R_{S'}$ such that $\boldsymbol{q} \cdot \boldsymbol{r}^{(d)} > 0$?

Give a proof if your answer is "yes" and a simple counter-example if your answer is "no".

(d) *[5 points]* The complement of an index set S is the the set of terms not in S, i.e., $\{1, \ldots, V\} - S$. Let $C(S)$ be the complement of S.

Use linearity of inner products to prove that for all $\boldsymbol{r} \in R$, and all index sets S,

$$\boldsymbol{q} \cdot \boldsymbol{r} \;\leq\; \boldsymbol{q}_S \cdot \boldsymbol{r} + \boldsymbol{q}_{C(S)} \cdot \boldsymbol{u}$$

(e) *[4 points]* Prove that for all $\boldsymbol{r} \in R$, and all $S \subseteq \text{support}(\boldsymbol{q})$,

$$\boldsymbol{q} \cdot \boldsymbol{r} \;\leq\; \boldsymbol{q}_S \cdot \boldsymbol{r} + \boldsymbol{q}_T \cdot \boldsymbol{u}$$

where $T = \text{support}(\boldsymbol{q}) - S$.

(f) *[3 points]* Based on the above and part (b), propose a simpler upper bound on $\boldsymbol{q} \cdot \boldsymbol{r}$ for $\boldsymbol{r} \notin R_S$ that holds for any index set $S \subseteq \text{support}(\boldsymbol{q})$.

(g) *[5 points]* Given a query $\boldsymbol{q}$ and a minimum similarity score ℓ, we want to find an index set S such that R_S contains all documents $\boldsymbol{r} \in R$ such that $\boldsymbol{q} \cdot \boldsymbol{r} \geq \ell$. We will use the following greedy algorithm to find a small index set S.

let $S_0 = \emptyset$; let $i = 1$
while *not_enough*$(S_i, \boldsymbol{u}, \boldsymbol{q}, \ell)$ is true
    $t_i = \text{argmax}_t$ *score*$(t, \boldsymbol{u}, \boldsymbol{q})$
    $S_i = S_{i-1} \cup \{t_i\}$
    $i = i + 1$

Complete the algorithm by defining efficient functions *not_enough* and *score*. Explain why *not_enough* is correct (i.e., when the loop stop R_{S_i} contains all the necessary documents) and why *score* is plausible (i.e., leads to smaller index sets.)

1.2 Combiners and the "last reducer" problem [20 Points]

A Map-Reduce step is not complete until every reducer is finished. One common performance problem occurs when the partitioning used by the Shuffle-Sort is uneven, and one reducer gets more than a fair share of items to process. This reducer will take longer to process its data, delaying completion of the whole job. This is sometimes called the "last reducer" problem.

As an example, suppose we ran the `wordcount` algorithm from class on a corpus with N terms and used R reducers. In an ideal partitioning, each reducer would get N/R messages to process. However, since all messages for the same word go to the same reducer, and some words are more frequent than others, an equal partitioning of *words* need not cause an equal partitioning of *word messages*.

(a) *[3 points]* Suppose the corpus contains only product offering documents. Each document is at most K words, and 90% of them contain the word "price". Give a lower bound on the total number of messages `('price', 1)` that will be sent.

(b) *[4 points]* Suppose $K = 50$. For what values of R can you be certain that at least one reducer will have more than N/R messages?

(c) *[3 points]* A *combiner* runs on the same machine as the mapper process which serves to compress the messages produced by the mapper before they are sent to the Shuffle-Sort. It is similar to a reducer, but will aggregate messages within a partition. The aggregated messages from each partition are then sent to the Shuffle-Sort these aggregated messages are then passed to each reducer. One way to use combiners in PySpark is with `combineByKey`.[1] An example of `wordcount` is:

```
word_counts = rdd.flatMap(lambda line:  line.split(" "))
                 .map(lambda word:  (word, 1))
                 .combineByKey(
                      lambda x:  x,          # convert value to accumulator
                      lambda x, y:  x + y,   # combine accumulator with a value, in-partition
                      lambda x, y:  x + y)   # combine accumulators, across partitions
```

Continuing this example, suppose the corpus is divided into P partitions. Give an *upper bound* on the total number of "price" messages that will be sent to the reducers by the Shuffle-Sort.

(d) *[5 points]* Assume the number of reducers R is the same as the number of partitions P, and that $K = 100$. Let n_0 be the number of "price" messages seen by the "price" reducer without combiners, and n_1 be the number of "price" messages seen by the "price" reducer with combiners.

For what values of P can you be certain that $n_1 < n_0$.

[1] The `fold` transformation also does mapper-side aggregation, looks more like `reduce`, but is less general.

(e) *[5 points]* Suppose instead the product offers are structured records, the price in dollars (as a `float`) can be extracted from a record with the function `getPrice` and the category (a string) can be extracted with `getCategory`. Complete the code below, which finds the average price for each product category.

```
def initial_pair(record):
    #convert a record to a pair (running_sum, count)
    return (getPrice(record), 1)
def in_partition_aggregation_fn(pair1, record):
    ...
def global_partition_aggregation_fn(pair1, pair2):
    ...
records = loadProductOfferRecords()
categoryPricePairs = records.map(
    lambda r:  (getCategory(r), r))
categoryPriceSumCountPairs = records.combineByKey(
    initial_pair_record,
    in_partition_aggregation_fn,
    global_aggregation_fn)
categoryAveragePrices = records.mapValues(
    lambda pair:  pair[0] / pair[1])
```

2 Programming [50 points]

2.1 Introduction

This assignment involves understanding some basics of distributed computing, the MapReduce programming model, Spark, and an example of data cleaning.

This assignment consists of two major parts. The first part is a tutorial on entity resolution, a common type of data cleaning. The second part is an implementation of PCA and application to a biological light-sheet imaging dataset.

We provide the code template for this assignment in *two* Jupyter notebooks. What you need to do is to follow the instructions in the notebooks and implement the missing parts marked with '`<FILL IN>`' or '`# YOUR CODE HERE`'. Most of the '`<FILL IN>/YOUR CODE HERE`' sections can be implemented in just one or two lines of code. Also, keep in mind to delete the '`raise NotImplementedError()` once you are done implementing a function. We provide several `assert` statements in the notebook for you to check the validity of your solution.

2.2 Getting started

2.2.1 Getting lab files

You can find the notebooks '`hw1_coding_1.ipynb`' and '`hw1_coding_2a.ipynb`' and '`hw1_coding_2b.ipynb`' in the homework 1 handout .tar file.

Next, import the notebooks into your Databricks account, which provides you a well-configured Spark environment and will definitely save your time (see the next section for details).

2.2.2 Databricks

We provide step-by-step instructions on how to configure your Databricks platform. We will also introduce it in detail during the recitation on January 17th.

1. Sign up for the **Community Edition** of Databricks here: https://databricks.com/try-databricks.
2. Import the notebook file we provide on your homepage: `Workspace -> Users -> Import`
3. Create a cluster: `Clusters -> Create Cluster`. You can use any cluster name as you like. When configuring your cluster, make sure to choose **runtime version** `14.3 LTS`. Note: It may take a while to launch the cluster, please wait for its status to turn to '`active`' before start running.
4. Installing third-party packages that will be used in the homework on Databricks: `Clusters -> Cluster name -> Libraries -> Install New`. Then select `PyPI`, enter the package name as `nose`. Finally click `Install` to install it.
5. You can start to play with the notebook now!

Note: Databricks Community Edition only allows you to launch one 'cluster'. If the current cluster is 'terminated', then you can either (1) delete it, and then create a new one, or (2) activate and attach to the existing cluster when running the notebook. Make sure to install nose.

2.2.3 Preparing for submission

We provide several public tests via `assert` in the notebook. You may want to pass all those tests before submitting your homework. You can individually submit a notebook for debugging but make sure to submit both notebooks for your final submission to receive full credit.

Important! In order to enable auto-grading, please do not change any function signatures (e.g., function name, parameters, etc) or delete any cells. If you do delete any of the provided cells (even if you re-add them), the autograder will fail to grade your homework. If you do this, you will need to re-download the homework files and fill in your answers again and resubmit.

Important! Before submission to Gradescope please make sure that the "Spark Cell" is uncommented, as seen below

```
# YOU CAN MOST LIKELY IGNORE THIS CELL. This is only of use for running this notebook
    locally.

# THIS CELL DOES NOT NEED TO BE RUN ON DATABRICKS.
# Note that Databricks already creates a SparkContext for you, so this cell can be skipped.
import pyspark
from pyspark.sql import SparkSession, SQLContext

spark = SparkSession.builder \
    .appName("hw") \
    .config("spark.ui.showConsoleProgress", "False") \
    .getOrCreate()

sc = spark.sparkContext
sc.setLogLevel("OFF")
sqlContext = SQLContext(sc)

print("spark context started")
```

Warning! The autograder has been known to time-out poorly optimized solutions. Please prepare to submit to the autograder at least 1 day in advance. As always, please start your homework early.

2.2.4 Submission

1. Export both solution notebooks as IPython notebook files on Databricks via `File -> Export -> IPython Notebook`
2. Submit both completed notebooks and deliverables via Gradescope (you can select both notebooks when uploading your solutions).

2.3 Instructions

2.3.1 Entity Resolution

Entity Resolution, or "Record linkage" is the term used by statisticians, epidemiologists, and historians, among others, to describe the process of joining records from one data source with another that describe the same entity. Other terms with the same meaning include, "entity disambiguation/linking", "duplicate detection", "deduplication", "record matching", "(reference) reconciliation", "object identification", "data/information integration", and "conflation".

Entity Resolution (ER) refers to the task of finding records in a dataset that refer to the same entity across different data sources (e.g., data files, books, websites, databases). ER is necessary when joining datasets based on entities that may or may not share a common identifier (e.g., database key, URI, National identification number), as the case may be due to differences in record shape, storage location, and/or curator style or preference. A dataset that has undergone ER may be referred to as being cross-linked. In this homework, we break the entity resolution problem into four parts:

- **Part 0 – Preliminaries:** Load in the dataset into pair RDDs where the key is the mapping ID, and the value is a string consisting of the name/title, description, and manufacturer of the corresponding record.
- **Part 1 – ER as Text Similarity - Bags of Words:** Build components for bag-of-words text analysis, and then compute record similarity. Bag-of-words is a conceptually simple yet powerful approach for text analysis. The idea is treating strings, a.k.a. documents, as unordered collections of words or tokens, i.e., as bags of words.
- **Part 2 – ER as Text Similarity - Weighted Bag-of-Words using TF-IDF:** In this part we compute the TF-IDF weight for each record. Bag-of-words comparisons do not perform well when all tokens are treated in the same way. In real world scenarios, some tokens are more important than the others. Weights give us a way to specify which tokens could have higher "importance". With weights, when we compare documents, instead of counting common tokens, we sum up the weights of common tokens. A good heuristic for assigning weights is called "Term-Frequency/Inverse-Document-Frequency," or TF-IDF for short. **TF** rewards tokens that appear many times in the same document. It is computed as the frequency of a token in a document. **IDF** rewards tokens that are rare overall in a dataset. The intuition is that it is more significant if two documents share a rare word than a common one.
- **Part 3 – ER as Text Similarity - Cosine Similarity:** Compute the cosine similarity of the tokenized strings based on the TF-IDF weights.
- **Part 4 – Scalable ER:** Use the inverted index data structure to scale ER. The ER algorithm above is quadratic in two ways. First, we did a lot of redundant computation of tokens and weights, since each record was reprocessed every time it was compared. Second, we made quadratically many token comparisons between records. In reality, most records have nothing (or very little) in common. Moreover, it is typical for a record in one dataset to have at most one duplicate record in the other dataset (this is the case assuming each dataset has been de-duplicated against itself). In this case, the output is linear in the size of the input and we can hope to achieve linear running time. An inverted index is a data structure that will allow us to avoid making quadratically many token comparisons. It maps each token in the dataset to the list of documents that contain the token. So, instead of comparing, record by record, each token to every other token to see if they match, we will use inverted indices to look up records that match on a particular token.

- **Part 5 − Analysis:** Determine duplicate entities based on the similarity scores, and compute evaluation metrics. Now we have an authoritative list of record-pair similarities, but we need a way to use those similarities to decide if two records are duplicates or not. The simplest approach is to pick a threshold. Different thresholds correspond to different false positives and false negatives, which will result in different precision and recall scores.

See the notebooks for detailed descriptions and instructions of each question.

2.3.2 Streaming Naive Bayes

Much of machine learning with big data involves - sometimes exclusively - counting events. Multinomial Naive Bayes fits nicely into this framework. The classifier needs just a few counters.

For this assignment we will be performing document classification using streaming Multinomial Naive Bayes. We call it streaming because we won't load the training data into memory: instead we will load one document at a time, use that document to update the statistics that define the classifier, and then discard the document. The streaming formulation allows us to process large amounts of data—more than can fit in memory.

Let y be the labels for the training documents and wi be the ith word in a document. Here are the counters we need to maintain:

(Y=y) for each label y the number of training instances of that class

(Y=*) here * means *anything*, so this is just the total number of training instances.

(Y=y,W=w) number of times token w appears in a document with label y.

(Y=y,W=*) total number of tokens for documents with label y.

The learning algorithm just increments counters:

```
for each example {y [w1,...,wN]}:
    increment #(Y=y) by 1
    increment #(Y=*) by 1
    for i=i to N:
        increment #(Y=y,W=wi) by 1
    increment #(Y=y,W=*) by N
```

Classification will take a new document with words $w_1, ..., w_n$ and score each possible label y with the natural log probability of y as in Equation 1. At classification time, use Laplace smoothing with $\alpha = 1$ as described here: http://en.wikipedia.org/wiki/Additive_smoothing. Therefore, you also need track the vocabulary size during training.

$$ln(p(Y = y)) + \sum_{w_i} ln(p(W = w_i|Y = y)) \tag{1}$$

***Important Notes*:**

- You may keep a hashtable(note: hashtables are implemented with 'dict' in python) in memory, with keys like "Y=news", "Y=sports,W=aardvark", etc.
- You may NOT load all the training documents in memory. That is, you must make one pass through the data to collect the count statistics you need to do classification.
- You may assume that all of the test documents will fit into memory

The RCV1 Data:

For this assignment, we are using the Reuters Corpus, which is a set of news stories split into a hierarchy of categories. There are multiple class labels per document. This means that there is more than one correct answer to the question "What kind of news article is this?" For this assignment, we will ignore all class labels except for those ending in CAT. This way, we'll just be classifying into the top-level nodes of the hierarchy:

- CCAT: Corporate/Industrial
- ECAT: Economics
- GCAT: Government/Social
- MCAT: Markets

There are some documents with more than one CAT label. Treat those documents as if you observed the same document once for each CAT label (that is, add to the counters for all labels ending in CAT). If you're interested, a description of the class hierarchy can be found at http://www.jmlr.org/papers/volume5/lewis04a/lewis04a.pdf.

The format is one document per line, with the class labels first (comma separated), a tab character, and then the document. There are three file sets:

```
RCV1.full.*
RCV1.small.*
RCV1.very_small.*
```

The two file sets with "small" in the name contain smaller subsamples of the full data set. They are provided for debugging and local tests. Each data set is split into a train and test set, as indicated by the file suffix.

Output Format:

Once you pass all the local tests in the notebook, you will implement Streaming Naive Bayes on the **full dataset** and write the classification results to a file, `full_result.txt`. The output format should have one test result per line, and each line should have the format:

[Label1, Label2, ...]<tab>Best Class<tab>Log prob

where **[Label1, Label2, ...]** are the true labels of the test instance, **Best Class** is the class with the maximum log probability (as in Equation 1), and the last field is the log probability. The last line of the file should include the accuracy. Here's the expected output of `very_small_test` dataset for your reference:

```
['C24', 'CCAT', 'M14', 'MCAT']    MCAT    -9893.7804
['E51', 'E512', 'ECAT', 'GCAT', 'GDIP']    ECAT    -3912.8180
['C15', 'C152', 'C18', 'C181', 'CCAT']    CCAT    -1121.5992
['GCAT']    ECAT    -1610.1660
['C13', 'CCAT', 'GCAT', 'GHEA']    CCAT    -701.3466
```

```
['C13', 'CCAT', 'M11', 'MCAT']    CCAT    -1453.3430
['C11', 'C13', 'CCAT', 'E12', 'ECAT', 'M13', 'M132', 'MCAT']    ECAT    -2218.3302
['C31', 'CCAT']    CCAT    -2285.0698
Accuracy: 7/8=0.8750
```

See the notebooks for detailed descriptions and instructions.

2.3.3 Naive Bayes on Spark

This assignment will involve you porting your Naive Bayes implementation from the previous section to Spark to run on Databricks. We are using a different dataset than the previous section. This dataset is extracted from DBpedia. The labels of the articles are based on the types of the document. There are in total 17 (16 + other) classes in the dataset, and they are from the first level class in DBpedia ontology.

The training and test data format is one document per line. Each line contains three columns which are separated by a single tab:

- a document id
- a comma separated list of class labels
- document words

You will implement key functions for the data pipeline, from data parsing to model evaluation. Much of the code has been provided to you, with areas that you are expected to fill in. The assignment is split up into 5 sections, containing sub-sections that you must complete. These are your deliverables:

1. Enviroment Setup
   (a) Pick your data sample
   (b) Parsing the raw data
2. Training the Naive Bayes Classifier
   (a) Compute vocabulary length
   (b) Compute the remainder of your model
3. Testing the model
   (a) Generating predictions
   (b) Checking accuracy
4. Top 10 words per label
5. Train and test on large dataset

Local test cases are provided that are meant to be run using the `tiny` data sample. Please ensure that when you are checking for correctness, you are using the correct data sample.

You will upload the completed notebook to Gradescope along with your `export.csv` file that you download in section 5. All necessary details to complete the assignment are included in the notebook.

2.4 Deliverables

The deliverable for the programming section are the completed `.ipynb` notebooks, `full_result.txt` for Naive Bayes and `export.csv` for Streaming Naive Bayes. Please submit them to the autograder on Gradescope. Your grade for the programming section will be entirely determined by the score given by the autograder.

Future programming assignments will have empirical and written reflection questions to be submitted in the handout with the written section, so please remember to check the deliverables in future homeworks!

3 Collaboration Questions

1. (a) Did you receive any help whatsoever from anyone in solving this assignment?

   (b) If you answered 'yes', give full details (e.g. "Jane Doe explained to me what is asked in Question 3.4")

2. (a) Did you give any help whatsoever to anyone in solving this assignment?

   (b) If you answered 'yes', give full details (e.g. "I pointed Joe Smith to section 2.3 since he didn't know how to proceed with Question 2")

3. (a) Did you find or come across code that implements any part of this assignment?

   (b) If you answered 'yes', give full details (book & page, URL & location within the page, etc.).