Project 2B Specification

Computer Science 143 - Spring 2019

Project 2 Part B

Due Sunday, 06/02/2019 by 11:59pm

Please start early.

Download the data: file 1, file 2, file 3. These files are hosted on an AWS Educate account with promotional credits, so if the files become unavailable, please contact us immediately.

Preface

1. This project is going to require a lot of Googling and StackOverflow work, which is an important skill to master. This simulates a project you may do outside of school. The only caveat is that if you *copy/paste* anything from StackOverflow etc, just cite it either in a comment, or in `README.txt`. If you do your research correctly, you may find this project to be easily manageable. This is a skill that takes practice, so there is no better time to start than now.
2. Unlike Project 1, Project 2 contains tasks and QUESTIONS. Please answer the questions in your final deliverable report.
3. If you have access to a server and you wish to install Spark, you can use that, but we cannot provide support and your code must run in the VirtualBox image. If you have **free** student credits on Amazon Web Services (AWS) or Google Cloud (GCP), you should feel free to use them, but again, we cannot provide support and your code must run on the VirtualBox image.

Project 2B consists of several tasks. These tasks are designed to keep you on track, and are how your professor approached the problem. You should try to stick to the tasks given in this specification. If you deviate, you may run into problems that then require you to backtrack and try another approach, so proceed with caution.

Now that we have warmed up by writing a text parsing function, we can now train a machine learning model that attempts to determine the sentiment of a particular Reddit Politics comment. But it is not that simple -- (an em dash) we must first transform the data into a structure that allows us to train such a model. 95% of machine learning, data science, whatever, is cleaning data. It is crucial to get it right or we might get a bad model. *For CS 143, we are not judging accuracy the accuracy/performance of the model. There are other classes for that.*

What is a Classifier?

A classifier is essentially a function *f* that maps a set of features *X* to a label *Y*. A *feature* is some attribute that we believe will help us predict label *Y*. We will use many kinds of features: unigrams, bigrams and trigrams. *Y*, the label (sometimes called the target) is some measure of sentiment. It can be {-1, +1} to represent negative and positive respectively. It can also be a vector, where each element in the vector is a probability that the comment is positive, and the probability that the comment is negative. We will train two classifiers for each dimension (more on this in a bit): one for positive, and one for negative. Why? Positive and negative are not necessarily opposites. The opposite of positive is not positive and the opposite of negative is not negative. If they were in fact opposites, we would use one classifier. It is our hope that a comment is either positive/not negative, negative/not positive, not positive/not negative. *f* can take on many forms. It can be linear, or nonlinear. If our goal were to predict continuous values, the simplest model *f* is linear regression from Stats 10. In classification, the simplest linear model is called *logistic regression*.

Some other forms of *f* include support vector machines (SVM), random forests, naive bayes (based on Bayes rule and conditional probability) and neural networks and the ever prescient "Deep Learning."

Training a classifier is as simple as fitting a model to the data. But we have to be careful what data we use. We will eventually want to know how accurate our classifier is. At first, we can train the model using some part of the data, and then evaluate accuracy on the same part of data. This is called the *training accuracy*, and gives us an indicator of whether or not we are even in the right ballpark of classifier choice. But we cannot rely on only the training accuracy. The sample of data on which we trained the classifier may not be representative of all of the data we may see. If we only use training accuracy, we may as well just have a classifier that memorizes the dataset. That is, the data do not form a random sample of all Reddit Politics comments.

Take this ridiculous extreme example. Suppose we train a model by simply memorizing the data and their associated sentiment labels. We can get 100% accuracy on this training set. But if we run unseen text through the classifier to make predictions, the accuracy will likely be terrible. So, we divide the data into two sets: a training set, and a testing set. We train, or fit, a model *f* on the training set. To measure accuracy, we then apply the classifier to the test set. Your professor tends to pick a 70/30 or 80/20 split on training and testing data. However, we do not have much labeled data, so we may take a different approach.

The Data

In 2018, your professor took a sample of 2,000 Reddit Politics comments and posted them to Mechanical Turk, a service where humans perform small human intelligence tasks for a small wage (10 cents is typical). In this task, the Turker read the comment, and then rated the sentiment of the comment (positive, negative, neutral, not applicable) on three dimensions:

1. sentiment towards Democrats,
2. sentiment towards President Trump,
3. and sentiment towards Republicans.

This year, we will only use #2.

Sentiment Survey Instructions (Click to expand)

You will be presented with a comment from Reddit /r/politics. Read each comment, understand what it is saying, and then answer the following questions regarding what sentiment (positive or negative) each comment conveys.

- If the comment is related to **Donald Trump or his administration**, choose **positive** if *overall* the comment expresses a positive emotion towards Trump, choose **negative** if *overall* the comment expresses a negative emotion towards Trump, choose **neutral** if the comment is *entirely* factual in nature or does not express any emotion towards Trump and choose ***Not Applicable*** if the comment has nothing to do with **Donald Trump or his administration**.
- If the comment is related to **Republicans,** either specific Republicans (including Trump), Republican/Conversative policy, or the Republican Party, choose **positive** if *overall* the comment expresses a positive emotion towards **Republicans**, choose **negative** if *overall* the comment expresses a negative emotion towards **Republicans**, choose **neutral** if the comment is *entirely* factual in nature or does not express any emotion towards Republicans and choose ***Not Applicable*** if the comment has nothing to do with **Republicans**.
- If the comment is related to **Democrats,** either specific Democrats (including Hillary Clinton), Democratic/Liberal/Progressive policy, or the Democratic Party, choose **positive** if *overall* the comment expresses a positive emotion towards **Democrats**, choose **negative** if *overall* the comment expresses a negative emotion towards Democrats, choose **neutral** if the comment is *entirely* factual in nature or does not express any emotion towards Democrats and choose ***Not Applicable*** if the comment has nothing to do with **Democrats**.

To receive payment, there must be a response to ALL questions.

Comment:

${text}

1. What sentiment is expressed about Donald Trump?

- Positive
- Negative
- Neutral/factual/no clear sentiment
- Not applicable

2. What sentiment is expressed about Republicans?

- Positive
- Negative
- Neutral/factual/no clear sentiment
- Not applicable

3. What sentiment is expressed about Democrats?

- Positive
- Negative
- Neutral/factual/no clear sentiment
- Not applicable

Each comment was shown to three different individuals, and the eventual "true" sentiment label for the comment was determined to be the most common response for each of the three dimensions. Unfortunately, Mechanical Turk generally yields very poor results unless the task specification is crystal clear, and the task is targeted to particular subpopulations.

The file `labeled_data.csv` contains a comment_id and sentiment labels on three dimensions (Democrat, Republican, Trump specifically).

1. -1 means negative
2. 1 means positive
3. 0 means neutral
4. -99 means not applicable

We will only work with positive and negative labels.

Getting the Data Ready for Training and Testing a Model

We need to gather our features. The three sets of features are the unigrams, bigrams and trigrams you computed with the `sanitize` function. We will now use this function to add to a Spark DataFrame.

Loading a BZ2 file containing JSON objects into Spark is easy:

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
sc.addPyFile("cleantext.py")
comments = sqlContext.read.json("comments-minimal.json.bz2")
submissions = sqlContext.read.json("submissions.json.bz2")
```

What is with the `addPyFile` call? Spark is a distributed framework. It is most powerful running on a cluster such as Mesos or Yarn, but for this class we will run Spark locally. Still, Spark *may* use additional cores on your machine by forking separate processes. These other processes may be forked off somewhere else that cannot access your `cleantext.py`. So, we tell Spark to add `cleantext.py` to a distributed cache, which is accessible to all nodes in the Spark cluster, even if it is local.

Data Management Tasks

You will want to read Spark documentation for these tasks:

1. Spark SQL and Data Frames
2. Spark SQL Guide (includes information about User Defined Functions)
3. Open Source Spark SQL Guide
4. Extracting and Constructing Features in Spark MLLib
5. There are two ways to work with the relational model in Spark: 1) use Spark primitives such as `join`, `select` etc., 2) use SQL with the `sql` function.

I M P O R T A N T: All tasks must be completed in one Python file called `reddit_model.py` (underscore, not hyphen) and your main entry point must be accessible via the `main` guard. Click here for an example of some Python code. Click here for the coding template.

N O T E: Spark also uses the word "tasks" to refer to a particular computation. The word "task" in this spec should not be confused with Spark's definition.

TASK 1: Load the comments (BZ2 JSON), submissions (BZ2 JSON) and labeled data (CSV) into PySpark. This may take a while. To prevent having to wait 10 mins each time the data are loaded, you can write it to your shared directory as a parquet file.

```
some_data_frame_name.write.parquet("somefilename")
```

and then load it quickly. You may need to import a library to get this to work though.

TASK 2: To train a classifier, we only need to work with the labeled data, but `labeled_data.csv` ONLY contains a `comment_id`, and 3 sentiment labels. The comments file contains the actual comments and a bunch of other information. We need to do something with these two data frames so we only have data associated with the labeled data.

QUESTION 1: Take a look at `labeled_data.csv`. Write the functional dependencies **implied** by the data.

QUESTION 2: Take a look at the schema for comments. Forget BCNF and 3NF. Does the data frame *look* normalized? In other words, is the data frame free of redundancies that might affect insert/update integrity? If not, how would we decompose it? Why do you believe the collector of the data stored it in this way?

~~**TASK 3:** We also need to add in features that denote which subreddits each user contributes to either via comment or via submission. We have this information in both BZ2 files.~~ We are only using text features this quarter.

TASK 4: Generate the unigrams, bigrams and trigrams for each comment in the labeled data and store all of them combined into one column. You have already written this function in project 2A, but we cannot just call it like we would for other Python code. You will need to write a UDF (user defined function) -- a special function that Spark understands.

TASK 5: To train a model, we must have all features in one column, as an array of strings. Combine the unigrams, bigrams, trigrams ~~and participating subreddits~~ into the same column. You may need to write a UDF for this.

That is, your sanitize function returns something like the following:

```
['a string of unigrams', 'a string of bigrams', 'a string of trigrams']
```

We want the following representation instead:

```
['a', 'string', 'of', 'unigrams', 'a', 'string', 'of', 'bigrams', 'a',
'string', 'of', 'trigrams']
```

You can either modify your sanitize function, or call your function as is and reorganize the return value so it matches the above.

For this task, you will want to review the Spark documentation for `CountVectorizer`.

TASK 6A: Use a binary `CountVectorizer` to turn the raw features into a sparse feature vector, a data structure that Spark ML understands. Only use tokens that appear more than 10 times across the entire dataset (the `minDf` parameter). A feature vector is a standardized way of representing which features are in a particular observation. Each feature may have a value associated with it such as a count or a TF-IDF value. For our purposes, it is just 0/1 (absent/present). Since the representation is sparse, you will only see 1s (as well as a bunch of integers that index tokens in the vocabulary). Taking advantage of sparseness is a great way to preserve RAM.

TASK 6B: Create two new columns representing the positive and negative labels. Take the original labels {1, 0, -1, -99} and do the following. Construct a column for the positive label that is 1 when the original label is 1, and 0 everywhere else. Construct a column for the negative label that is 1 when the original label is -1 and 0 everywhere else.

Machine learning requires a TON of data management, so careers in these fields (or data science, whatever) requires strong understanding of relational databases, whether you use SQL in Spark, or Spark natives.

Model Fitting Tasks

Now it is time to fit a model! Since this is a Databases class and not a Machine Learning class, we will focus more on Spark and working "big data" (though this is far from big data on a VM... but the principle remains the same if we use a cluster with TBs of data). We will use a model called Logistic Regression, which is similar to Linear Regression you learned in Stats 10.

In linear regression we have,

$$E\left[y|x\right] = \beta_0 + \beta_1 x_1 + \ldots + \beta_k x_k$$

where the left hand side just represents an average prediction of *y* given data *x*. In logistic regression, we have

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \ldots + \beta_k x_k$$

where the left-hand side is called the *logit*. Given data *x,* we can predict the probability that *y* = 1 (if *y* represents positivity, then *y* = *1* means positive, but if *y* represents negative sentiment, then y = 1 means negative), as

$$P(Y=1) = \frac{1}{1+e^{\beta_0+\beta_1 x_1+\ldots+\beta_k x_k}}$$

where the betas are estimated using some optimization algorithm such as maximum likelihood estimation, Newton-Raphson Method, or Stochastic Gradient Descent, and *x* is the data.

TASK 7: Train a logistic regression model on the features (unigrams, bigrams and trigrams). Instead of dividing the data into a training and testing set, we will use a method called *k* fold cross-validation.

80% Training	20% Testing

With *k* fold cross-validation, we divide the data into *k* non-overlapping parts. At each iteration, we pick one of the parts (folds) to use as the testing set (20% for 5 folds) and we use the rest (80% for 5 folds) to train the model. In the next iteration, we choose a new testing fold, and so on. This yields *k* models, and *k* sets of accuracy metrics. The "overall" accuracy or performance of a model is usually the *average* performance across the *k* iterations.

Iteration 1	20% Train	20% Train	20% Train	20% Train	20% **Test**	80% Train, 20% Test
Iteration 2	20% Train	20% Train	20% Train	20% **Test**	20% Train	80% Train, 20% Test
Iteration 3	20% Train	20% Train	20% **Test**	20% Train	20% Train	80% Train, 20% Test
Iteration 4	20% Train	20% **Test**	20% Train	20% Train	20% Train	80% Train, 20% Test
Iteration 5	20% **Test**	20% Train	20% Train	20% Train	20% Train	80% Train, 20% Test

We typically use *k* fold cross-validation when we are not working with a lot of training data (our case here), or when we have reason to suspect that a simple train/test split will yield biased results.

Since this is not a machine learning class, you can use the following code to train the positive and negative models for Trump.

You will want to walk to the farthest away Starbucks to get a cup of coffee, because this process may take a while.

```
# Bunch of imports (may need more)
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Initialize two logistic regression models.
# Replace labelCol with the column containing the label, and featuresCol with
the column containing the features.
poslr = LogisticRegression(labelCol="poslabel", featuresCol="features",
maxIter=10)
neglr = LogisticRegression(labelCol="neglabel", featuresCol="features",
maxIter=10)
# This is a binary classifier so we need an evaluator that knows how to deal
with binary classifiers.
posEvaluator = BinaryClassificationEvaluator()
negEvaluator = BinaryClassificationEvaluator()
# There are a few parameters associated with logistic regression. We do not
know what they are a priori.
# We do a grid search to find the best parameters. We can replace [1.0] with a
list of values to try.
# We will assume the parameter is 1.0. Grid search takes forever.
posParamGrid = ParamGridBuilder().addGrid(poslr.regParam, [1.0]).build()
negParamGrid = ParamGridBuilder().addGrid(neglr.regParam, [1.0]).build()
# We initialize a 5 fold cross-validation pipeline.
posCrossval = CrossValidator(
    estimator=poslr,
    evaluator=posEvaluator,
    estimatorParamMaps=posParamGrid,
    numFolds=5)
negCrossval = CrossValidator(
    estimator=neglr,
    evaluator=negEvaluator,
    estimatorParamMaps=negParamGrid,
    numFolds=5)
# Although crossvalidation creates its own train/test sets for
# tuning, we still need a labeled test set, because it is not
# accessible from the crossvalidator (argh!)
# Split the data 50/50
posTrain, posTest = pos.randomSplit([0.5, 0.5])
negTrain, negTest = neg.randomSplit([0.5, 0.5])
# Train the models
print("Training positive classifier...")
posModel = posCrossval.fit(posTrain)
print("Training negative classifier...")
```

```
negModel = negCrossval.fit(negTrain)

# Once we train the models, we don't want to do it again. We can save the
models and load them again later.
posModel.save("project2/pos.model")
negModel.save("project2/neg.model")
```

You may find a better way to write this code, but this is a quick template to get you going.

For those that care about machine learning: The grid search attempts to find the best regularization parameter. We won't worry about what that means, but for those that care: to figure out the values for the betas, we must minimize the classification error as described by a *loss function*. This is similar to an objective function as taught in CS 161. For logistic regression, the loss function is the logistic loss. But many times, just minimizing the loss is not enough. We sometimes also want to penalize models that have too much features (as we do here), or constrain wildly large weights. Left alone, these models are overly complex and tend to not generalize well to new data. We can choose to penalize overly complex models by increasing their loss using a regularization term represented by lambda.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

where $y^{(i)}$ represents a true value for y (0 or 1) and $h(x^{(i)})$ is the predicted value of y, given data x, and theta is just beta for this model. Lambda is the regularization parameter and *m* is the number of data points. Ok, enough of that...

Model Evaluation Task

CAUTION! In these tasks, you need to be very, very careful how you set up your joins. Remember what we learned in Chapter 12. You may also need to read documentation about how joins are implemented in Spark. If you receive `OutOfMemoryError`, it means you did not structure your join correctly with respect to Spark. When you take advantage of the features of a package like Spark, it is amazing how much data you can work with on a limited system. Hint: Check the ordering of a sample of contiguous rows. You can manually simulate a merge join, and Spark has a way of doing an indexed join. Or is there something even simpler we can do?

IF AFTER YOU'VE TRIED EVERYTHING, YOU STILL GET OUTOFMEMORY ERRORS, USE A SAMPLE OF THE DATAFRAME, TRY 50% AND IF THAT DOESN'T WORK, USE 20% FROM THIS POINT ON. Write in your `README.txt` what you did.

Task 8: You will now read in the full file `comments-minimal.json.bz2`. For the final deliverable, we are going to need a few more things:

1. The timestamp when the comment was created.
2. The title of the submission (post) that the comment was made on.
3. The state that the commenter is from.

#2 can be computed with a join on `link_id` (comments) and `id` (submissions). But what about #3? Look again at the schema in the Project 2A Specification. There is a field called `author_flair_text` that sometimes contains the name of a state that the commenter votes in. You can use this as the State.

A special note on link_id: the first three characters represent the type of link and looks like "`t3_`". This must be stripped off or your join will be the empty set.

You may want to restrict your data frame to only these columns, the comment ID, and the comment text to save space.

Task 9: Run your classifiers on this text. You will need to repeat Tasks 4, 5 and 6A (except with the CountVectorizer. Once you fit the CountVectorizer you don't need to fit it again. Use the transform method to apply the CountVectorizer to the unseen data) so that the format of the new unseen data is identical to the format of the training data with a few exceptions: (1) remove any comments that contain "/s" (denoting that the comment is sarcastic... machine learning does not deal with sarcasm well, and (2) remove any comments that start with > this means the comment contains a quote of another comment and likely contains multiple sentiments not just one. This will give you the probability that the comment is positive sentiment and a probability that the comment is negative sentiment. Then, we must threshold these probabilities to convert them into 0/1 indicating if the comment is positive sentiment and if the comment is negative sentiment (probabilities are not intuitive to work with). If the positive probability is greater than 0.2, label it the positive sentiment a 1. If the negative probability is greater than 0.25, label the negative sentiment as 1. These values were derived from the ROC curve, which is a visual diagnostic of how accurate a classifier is. While most may assume that the probabilities should be greater than 0.5 (a coin flip), this data is imbalanced, that is, the number of positive labeled comments is not the same as the number of negative labeled comments. Instead, we use this proportion as the cutoff, to represent a result that is obtained solely by chance, but enough of that. **You are free to pick your own cutoff, but this typically requires looking at the ROC curve (or doing a grid search over all possible cutoffs).**

Construct two new columns "pos" and "neg" that contain 0 or 1 based on the thresholding. It is possible that a comment has a 0 for both, or a 1 for both. So don't worry about that.

Below is an example of how to apply the classifier to unseen data to get predicted probabilities. You will then convert these probabilities to 0/1. You may have to play with this:

```
posResult = posModel.transform(dataframe_task9)
negResult = negModel.transform(dataframe_task9)
```

This will give you three new columns in posResult and negResult. One of them is called `probability` and is a vector of doubles and index 1 is the probability that a comment is positive in the positive result, and negative in the negative result. You will convert this value to 0/1 based on the thresholds I gave above. There is also `rawPrediction` and `prediction`. Unfortunately, it is not clear what threshold these predictions use so we will ignore them. Unless you used setThreshold, then it is based on that threshold. You can either manually convert the probabilities into labels, or you can add use `setThreshold()` on the `LogisticRegression` object .

Task 10: Perform the following computations:

1. Compute the percentage of comments that were positive and the percentage of comments that were negative across all submissions/posts. You will want to do this in Spark.
2. Compute the percentage of comments that were positive and the percentage of comments that were negative across all days. Check out from `from_unixtime` function.
3. Compute the percentage of comments that were positive and the percentage of comments that were negative across all states. There is a Python list of US States here. Just copy and paste it.
4. Compute the percentage of comments that were positive and the percentage of comments that were negative by comment and story score, independently. You will want to be careful about quotes. Check out the `quoteAll` option.

Store your results in dataframes and write them to disk.

Final Deliverable

This final deliverable will be a mini-report of your findings. You will take the data frames you wrote to disk (CSV, tab delimited, whatever makes sense) and import them either into Python (or R) and create a few plots for us. R and Python are not necessarily memory efficient, so we suggest sampling the final data frame (if you haven't already), and loading that sampled data frame into R or Python instead.

Since Spark is a distributed system that partitions data onto different nodes (in a cluster at least), writing a single CSV file is not trivial. But we can use a special driver to do it, by first repartitioning the data to only contain one partition. This will create a directory containing one CSV file (and a file called _SUCCESS).

```
myCoolDataFrame.repartition(1).write.format("com.databricks.spark.csv").option("header", "true").save("myCoolData.csv")
```

Then produce the following in your `report.pdf`. If you like R, use this code to generate the plots. It may need to be modified for your data files. If you prefer Python, use this example code for the plots. This code may need to be adapted for use with this year's course offering.

1. Create a time series plot (by day) of positive and negative sentiment. This plot should contain two lines, one for positive and one for negative. It must have data as an X axis and the percentage of comments classified as each sentiment on the Y axis.
2. Create 2 maps of the United States: one for positive sentiment and one for negative sentiment. Color the states by the percentage.
3. Create a third map of the United States that computes the *difference:* %Positive - %Negative.
4. Give a list of the top 10 positive stories (have the highest percentage of positive comments) and the top 10 negative stories (have the highest percentage of negative comments). This is easier to do in Spark.
5. Create TWO scatterplots where the X axis is the submission score, and a second where the X axis is the comment score, and the Y access is the percentage positive and negative. Use two different colors for positive and negative. This allows us to determine if submission score, or comment score can be used as a feature.
6. Write a paragraph summarizing your findings. What does /r/politics think about President Trump? Does this vary by state? Over time? By story/submission?

You must also answer the following questions in your report:

QUESTION 1: Take a look at `labeled_data.csv`. Write the functional dependencies implied by the data.

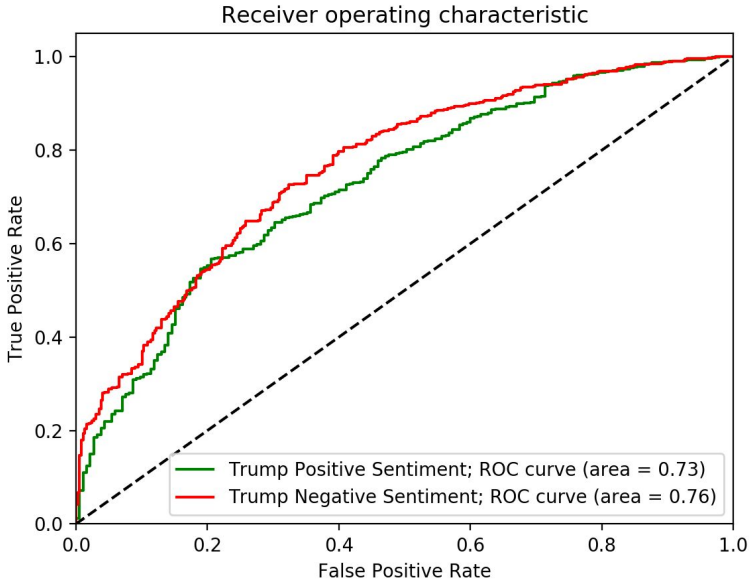
QUESTION 2: Take a look at the schema for the comments dataframe. Forget BCNF and 3NF. Does the data frame *look* normalized? In other words, is the data frame free of redundancies that might affect insert/update integrity? If not, how would we decompose it? Why do you believe the collector of the data stored it in this way?

QUESTION 3: Pick one of the joins that you executed for this project. Rerun the join with `.explain()` attached to it. Include the output. What do you notice? Explain what Spark SQL is doing during the join. Which join algorithm does Spark seem to be using?

We will be grading your code and execution, so not only is copying someone else's images/results prohibited, it won't work.

How Accurate is Accurate?

Since this is not a machine learning class, our main focus is ob data management and using the relational model (originally from JSON) rather than training a theoretically correct and accurate classifier, which is why I gave you the code. But, if you're curious, R3 got an AUC of 0.73 for the positive classifier and 0.76 for the negative classifier. In academia, these scores as so-so, but in real life, this is considered "fairly good."Professor was pleasantly impressed that such messy data, and such messy labels (Mechanical Turk is not usually reliable) produced such a good result.
Can you do even better? If so, let us know in your final deliverable report and describe what you did!



Submission Instructions

Preparing Your Submission

This project contained multiple TASKS. Please clearly denote each task using the following comment style. Your comment MUST match the style below. This tells us which tasks you completed.

```
# TASK 1
# Code for task 1…
```

Use this style *only* if you had to combine code for multiple tasks. Do not simply write one comment that mentions all tasks!

```
# TASKS 6A, 6B
# Code for tasks 6A and 6B
```

Please create a folder named with your UID, put all your files into the folder, then compress this folder into a single zip file called "P2B.zip". That is, the zip file should have the following structure.

```
P2B.zip
 |
 +- Folder named with Your UID, like "904200000" (without quotes)
     |
     +- README.txt
     |
     +- team.txt
     |
     +- reddit_model.py
     |
```

```
+- cleantext.py (p2a)
|
+- analysis.R or analysis.py
|
+- report.pdf (final deliverable, answers to our questions)
```

Please note that the file names are case sensitive, so you should use the exact same cases for the file names. (For teamwork, only the submitter's UID is needed to name the folder.) Here is more detailed description of each file to be included in the zip file:

- `README.txt`: Readme File mentioning anything necessary.
- `team.txt`: A plain-text file (no word or PDF, please) that contains the UID(s) of every member of your team. If you work alone, just write your UID (e.g. 904200000).
  If you work with a partner or a group, write all UIDs separated by a comma (e.g. 904200000, 904200001). Do not include any other content in this file!
- `reddit_model.py`: The file containing your Spark job.
- `analysis.R` or `analysis.py`: File containing code for your graphics (we don't need the data itself).
- `report.pdf`: Your report, answers to our questions, etc.

Submitting Your Zip File

Visit the Project 2B submission page on CCLE to submit your zip file electronically by the deadline. Submit only the "P2B.zip" file! In order to accommodate the last minute snafu during submission, you will have 30-minute window after the deadline to finish your submission process. That is, as long as you start your submission before the deadline and complete within 30 minutes after the deadline, we won't deduct your grade period without any penalty.

Real Life Lessons

This project is similar to one you might pursue if you were to be a data engineer or data scientist, but there are a lot of real-life lessons that you hopefully learned from doing this project that are highly applicable to software engineering:

1. It is important to save intermediate results. Data can take a while to load, and having to reload it several times a day is inefficient. Instead, save it in an efficient format (we used Parquet here) so it is easier to load back. This trick also allows you to begin development where you left off instead of at the beginning.
2. The power of sampling. Developing a minimum working example is an important skill. During development, the same is true. Instead of loading a 10TB file, and running your code on the 10TB file every single time you change something, use a small 1MB file until you are nearly certain it works properly. *This will save you hours or days of time*.
3. Set parameters as low as possible. In my model, I used a 10 fold cross validation. This took forever. While writing my code, I use a 2 fold cross validation, which yielded about a 5x speedup. Each time I failed, I only wasted 1 unit of time instead of 5 units of time.
4. The world is vague. Sometimes you just need to experiment with different methods to find a method that works. Books and your manager/professor won't always have the answer for you.

5. Learn how to read documentation and find information quickly. Reading documentation is important, but perhaps even more important is to be able to Google for answers to your questions *and identify the salient points quickly*. *That is exactly how I did this project and it is expected in industry... nobody knows everything.* In an academic environment, we ask you to cite any sources you use though just because it's what we do.
6. You have trained your first machine learning classifier. Some of you have already done this before, but I do not expect anyone to have done it before. If this was your first time, congratulations! I hope you continue.
7. Visualization is an easy way to communicate findings whether findings from data, or simplifying a complex concept to a non-technical or brand new audience.
8. The conclusions made in academia and in industry are different. Academia is based on theory, which usually yields the "best case scenario." We are taught that AUCs, accuracies, whatever must be as high as possible. *But this is not practical.* It is often the case that results that academia deems as "meh" are actually quite good for industry because contexts are much more complicated than they are in some toy example.
9. Fail fast, so the rollback is less painful. Creating an ETL job or a regular software program can be visualized either as a linear process, or a tree-like process. The farther down the line (or down the tree) you get, the more difficult it is to fix issues earlier on in the process. Once you fix an issue, you may then have to fix the code after it so it is compatible with your bugfix. Failing early takes a lot of experience. You should experiment with one feature or operation at a time. You will eventually get good at understanding if what you are implementing is sound, or if it was just a patch job that will break farther down the line. This is part of good software design. If you fail early, all you need to do is rollback the one thing that failed, fix it, and then move along on your merry way. If you fail late, you may not only have to fix one operation in question, but several operations/assumptions that came before it. Writing unit tests for each operation will prevent you from making changes that will break later down the line.

(Good unit tests can usually save you from 2, 3, and 9.)