Political Bias Article Classification

A KNN Algorithm Approach

By: Grant Baer, Logan Baker, Blake Silton and Kassi Winter

Table of contents

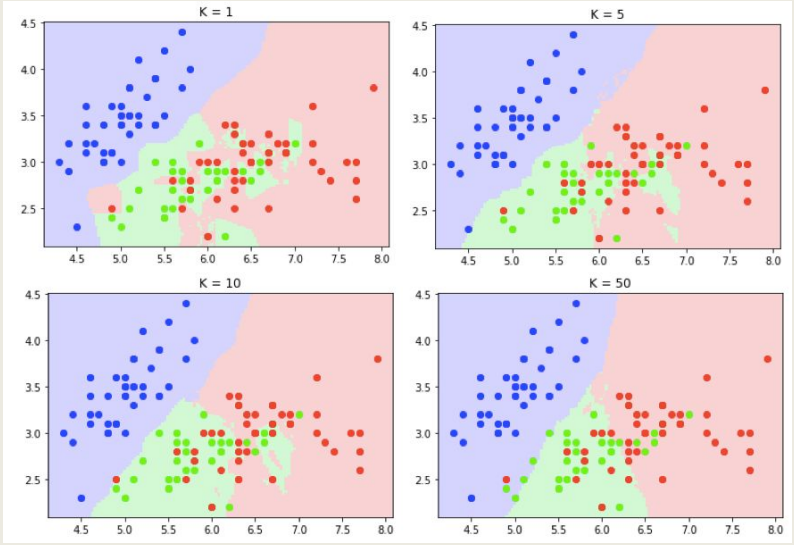
01

Project Idea

Getting Started

Acquired a comprehensive dataset of *political articles* labeled as 'left', 'center', and 'right'.

We chose to implement **K-Nearest Neighbors (KNN)** due to its effectiveness in handling *distinct group classifications*, aligning with our goal of *categorizing articles* based on political bias.



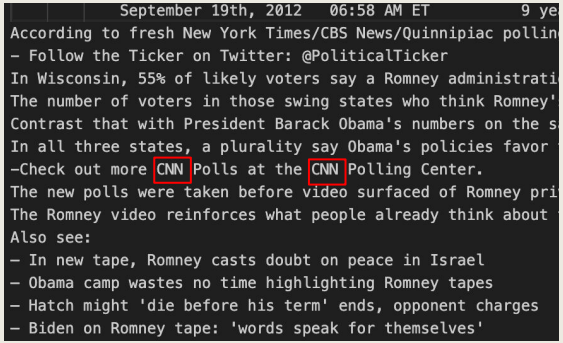
Step #1 Data Preprocessing

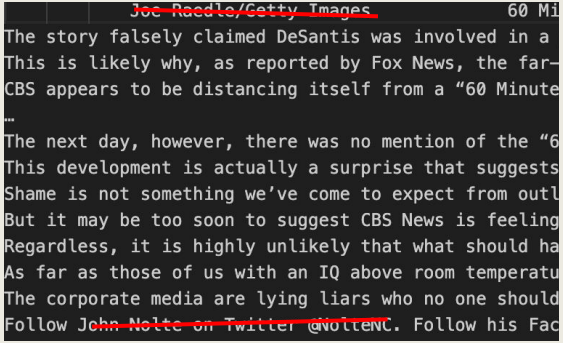
Removed all **non-alphabetical** characters from the raw articles.

Removed **common filler words** and other noise text from the articles.

Due to an imbalance in document numbers across categories, we meticulously **balanced the datasets** to ensure proportional representation across splits.

Normalized and **vectorized** each document, converting the text into numerical values using *Term Frequency-Inverse Document Frequency* (TF-IDF)

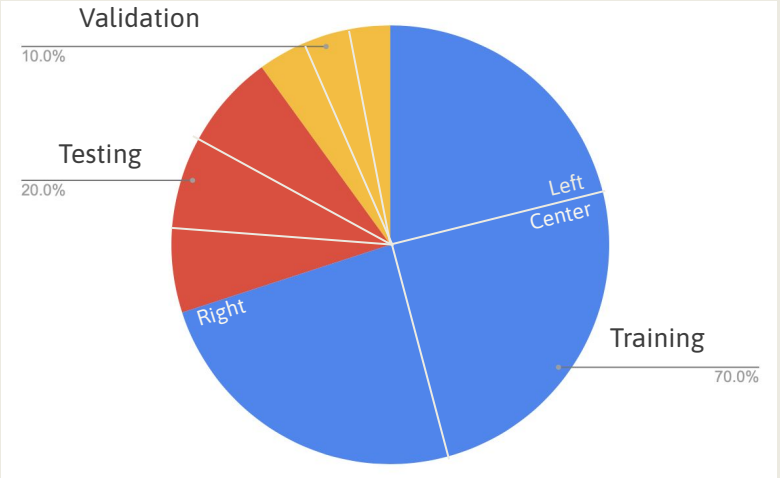




Step #2 Data Preparation

Split the data into document collections:

- Training (70%)
- Validation (10%)
- Testing (20%)



Encoded Actual Label into each document**:**

- -1 (Left)
- 0 (Center)
- 1 (Right)

Step #3 KNN Implementation

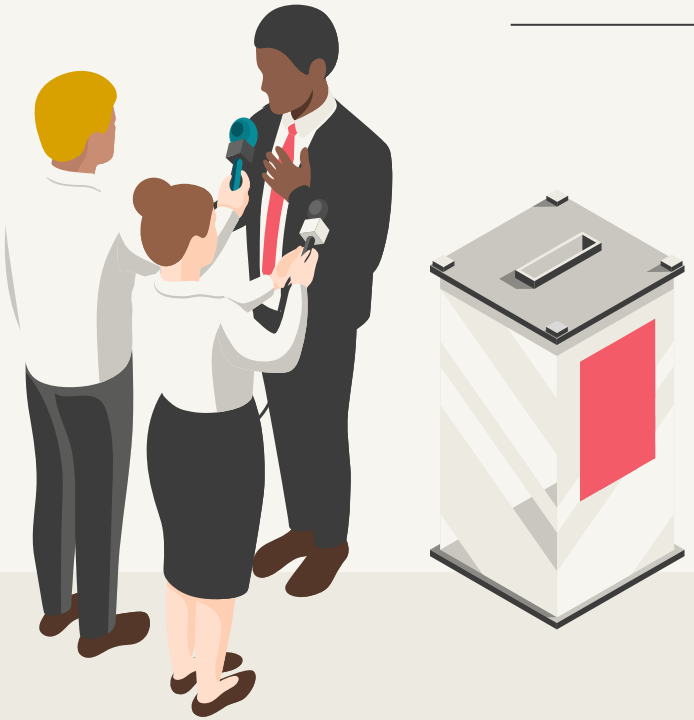
Nearest neighbors are determined by the **cosine distance** between document text vectors, post-normalization. We chose to use this calculation due to its effectiveness in high dimensional spaces (ignores of *document length*, *raw word count*, etc)

cosine similarity = $A \cdot B / ||A|| \cdot ||B||$

Our implementation is a true **KNN classifier**, since we *take the majority label* of the nearest neighbors to determine how to classify a novel sample.

02

How It Works



Project Structure

```
└─ src/
   ├─ DataProcessing/
   │   └─ DataProcessor.java
   ├─ DocumentClasses/
   │   ├─ CosineDistance.java
   │   ├─ DocumentCollection.java
   │   ├─ <interface> DocumentDistance.java
   │   ├─ DocumentVector.java
   │   ├─ <abstract> TextVector.java
   │   └─ files/
   │       └─ stopwords-en.txt
   └─ main/
       ├─ ArticleClassification.java
       └─ KNearestNeighbors.java
```

Class Overview

Data Processing → Cleans our dataset for use in KNN.

TextVector → Data model for article samples.

DocumentCollection → Stores TextVectors and provides useful methods for sample set manipulation.

KNearestNeighbors → Houses KNN logic. Finds an optimal k, predicts novel samples, tests results.

ArticleClassification → Main function. Assembles sample sets and drives use of KNearestNeighbors.

DataProcessing

`private static boolean verifyFileStructure()`: Ensures that the directory structure exists to store our processed data.

```
data/
├── raw/
│   ├── left/
│   ├── center/
│   └── right/
└── processed/
    ├── left/
    ├── center/
    └── right/
```

```java
private static boolean verifyFileStructure() {
    Path rawDataDir = Paths.get(rawDataDirPath);
    Path processedDataDir = Paths.get(processedDataDirPath);
    if (Files.notExists(rawDataDir)) {
        System.out.println(x:"Couldn't find raw data directory");
        return false;
    }
    for (String dataSubDir : dataSubDirs) {
        Path subDirPath = rawDataDir.resolve(dataSubDir);
        if (Files.notExists(subDirPath)) {
            System.out.println("Couldn't find raw data subdirectory " + subDirPath);
            return false;
        }
    }
    if (Files.notExists(processedDataDir)) {
        System.out.println("Couldn't find processed directory " + processedDataDir);
        if (!createDir(processedDataDir)) {
            return false;  // had an issue creating dir
        }
    }
    for (String dataSubDir : dataSubDirs) {
        Path subDirPath = processedDataDir.resolve(dataSubDir);
        if (Files.notExists(subDirPath)) {
            System.out.println("Couldn't find processed data subdirectory " + subDirPath);
            if (!createDir(subDirPath)) {
                return false;  // had an issue creating subdir
            }
        }
    }
    return true;
}
```

Convert raw data into all-lowercase, pruning any non-alphabetical characters.

Noise words are selectively excluded by TextVector when it is being

TextVector

Added a label attribute that gets stored based on the name of the parent directory of the file, i.e. if the file is in the center directory, it's label should be "`center`".

```java
public abstract class TextVector implements Serializable {

    /**
     * Stores the frequency for each non-noise word
     */
    private HashMap<String, Integer> rawVector;
    private Bias label;

    public TextVector(String category) {
        this.rawVector = new HashMap<>();
        switch (category) {
            case "left":
                this.label = Bias.LEFT;
                break;
            case "center":
                this.label = Bias.CENTER;
                break;
            case "right":
                this.label = Bias.RIGHT;
                break;
            case null, default:
                throw new IllegalArgumentException("Invalid article category: " + category);
        }
    }
```

DocumentCollection

- Modified the constructor to loop through files in a directory, and either:
  - take a `label` argument with which to initialize the `TextVector`, or
  - initialize each `TextVector` with the name of the directory.

- Changed constructor to only add non-stopwords to each `TextVector`.
  - Stop words are handled by `TextVector`

- Added a method to combine `DocumentCollection` instances.
  Doing this instead of iterating through multiple directories in the constructor reduces complexity & makes code more understandable.

KNearestNeighbors

`private int tuneK(double threshold)`: Choose an arbitrary starting value of K and run kNN on the documents in the validation set to compute accuracy. Increase K and repeat process from there until the accuracy is below threshold.

`public double[] test(int k)`: For every document in the testing set, run predict with k to get f1 accuracy score

`private int predict(TextVector sample, int k)`: Return the majority label of the k most similar documents to the given testing sample

```
public class KNearestNeighbors {
    public static final int PRECISION = 0;
    public static final int RECALL = 1;
    public static final int F1 = 2;
    /**
     * The number of closestDocuments to store under the hood. Saves repeat comput
     */
    private static final int numClosestDocuments = 3000;

    private final DocumentCollection trainingSet;
    private final DocumentCollection validationSet;
    private final DocumentCollection testingSet;

    /**…
    private final Map<TextVector, ArrayList<Integer>> closestTrainingDocs;
    private final HashMap<Integer, Map<Integer, TextVector>> trainingDocsByLabel;

    /**…
    public KNearestNeighbors(DocumentCollection trainingSet,…

    /**…
    public int tuneK(double threshold, int maxK) {…


    /**…
    public double[] test(int k) {…

    /**…
    private int predict(TextVector document, int k) {…
```

KNearestNeighbors

`private double[] calcPrecisionAndRecall(Hashmap<Integer, DocumentCollection> computerJudgement)`: Calculate and return the precision and recall of the returned computer results using the macro average formula

`private double calcF1(double precision, double recall)`: returns the calculated F1 Score

```
private double[] calcPrecisionAndRecall(HashMap<Integer, DocumentCollection> computerJudgement) {
    double totalPrecision = 0.0;
    double totalRecall = 0.0;
    HashMap<Integer, Integer> humanJudgment = new HashMap<>();
    System.out.println(x:"Computer Judgment");
    computerJudgement.forEach((key, value) -> System.out.println("Label: " + key + " Size: " + value.g
    // Loop #1: Get the number of actual given labels in each category
     for (int label : computerJudgement.keySet()) {
        DocumentCollection predictedLabelCollection = computerJudgement.get(label);
        Collection<TextVector> predictions = predictedLabelCollection.getDocuments();

        for (TextVector doc : predictions) {
            if (!humanJudgment.containsKey(doc.getLabel())) {
                humanJudgment.put(doc.getLabel(), value:0);
            }
            int newLabelCount = hu  int DocumentClasses.TextVector.getLabel()
            humanJudgment.put(doc.getLabel(), newLabelCount);
        }
    }
     System.out.println(x:"Human Judgment");
     humanJudgment.forEach((label, count) -> System.out.println("Label: " + label + " Count: " + count

    // Loop # 2: Get the number of correct label predictions (assigned vs accurate)
    for (int label : computerJudgement.keySet()) {                                    // go through
        DocumentCollection predictedLabelCollection = computerJudgement.get(label);
        Collection<TextVector> predictions = predictedLabelCollection.getDocuments();   // access all

        int numCorrect = 0;
        int numInCluster = predictions.size();
```

ArticleClassification

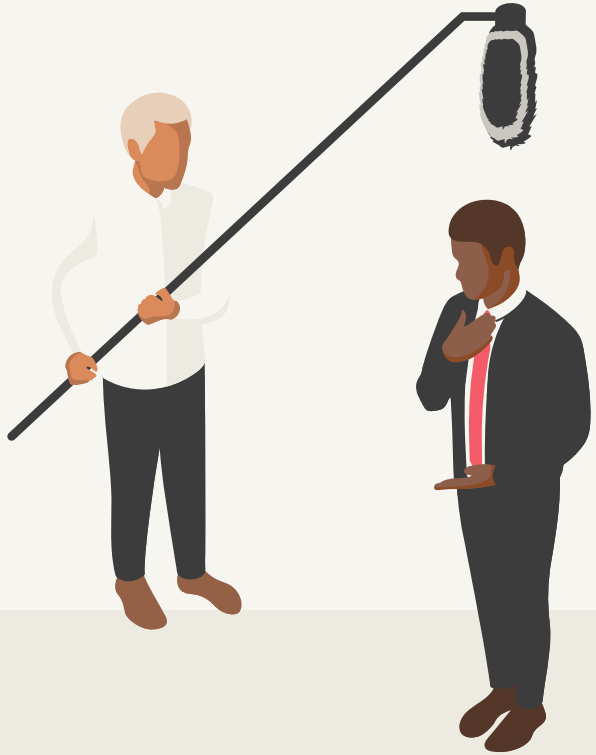
`main ()`: The main function containing all methods

`private static ArrayList<ArrayList<Integer>> getSetIDs(DocumentCollection data, double percentVal, double percentTest)` : Given a document collection representing a category of our data, i.e. left labeled articles, return array lists of IDs in the document collection representing documents to be included in the validation set and testing set respectively

```
public static void main(String[] args) {
    // going into main function, we have processed data for each category at ../data/processed
    HashMap<String, DocumentCollection> labeledDocCollections = new HashMap<>();
    try (DirectoryStream<Path> dataDirStream = Files.newDirectoryStream(Paths.get(dataDirPath),
            /* stream filter: */ path -> labels.contains(path.getFileName().toString()))) {…
    } catch (IOException e) {…

    // Attempt to load testing/training/validation sets from serialized objects. (Blake)
    System.out.println(x:"Attempting to read testing/training/validation sets...");
    DocumentCollection trainingSet = readDocumentCollection(serializedTrainingSetPath);
    DocumentCollection testingSet = readDocumentCollection(serializedTestingSetPath);
    DocumentCollection validationSet = readDocumentCollection(serializedValidationSetPath);

    // If any set couldn't be loaded, new sets must be constructed.
    if (trainingSet == null || testingSet == null || validationSet == null) {
        System.out.println("Failed to read serialized training, testing, or validation set.\n" +
                "Generating new DocumentCollections...");
        trainingSet = new DocumentCollection();
        validationSet = new DocumentCollection();
        testingSet = new DocumentCollection();
        ArrayList<ArrayList<Integer>> splits;

        for (DocumentCollection docCollection : labeledDocCollections.values()) {
            splits = getSetIDs(docCollection, percentVal:.1, percentTest:.2);

            for (int i = 1; i <= docCollection.getSize(); i++) {
                if (!splits.getFirst().contains(i) && !splits.getLast().contains(i)) {
                    trainingSet.addDocument(docCollection.getDocumentById(i));
                }
            }
            for (int index : splits.getFirst()) {
                validationSet.addDocument(docCollection.getDocumentById(index));
            }
            for (int index : splits.getLast()) {
                testingSet.addDocument(docCollection.getDocumentById(index));
            }
        }
        System.out.println(x:"Normalizing training set...");
        trainingSet.normalize(trainingSet, trainingSet.getSize(), set:"training");
        System.out.println(x:"Normalizing testing set...");
        validationSet.normalize(trainingSet, validationSet.getSize(), set:"validation");
        System.out.println(x:"Normalizing validation set...");
        testingSet.normalize(trainingSet, testingSet.getSize(), set:"testing");

        //Serialize generated sets (Blake)
        System.out.println(x:"Serializing normalized training set...");
        writeDocumentCollection(trainingSet, serializedTrainingSetPath);
        System.out.println(x:"Serializing normalized testing set...");
        writeDocumentCollection(testingSet, serializedTestingSetPath);
        System.out.println(x:"Serializing normalized validation set...");
        writeDocumentCollection(validationSet, serializedValidationSetPath);
    }
    double threshold = 0.0001;
    int maxK = 1000;
    KNearestNeighbors knn = new KNearestNeighbors(trainingSet, validationSet, testingSet);
    System.out.println(x:"Tuning k...");
    int k = knn.tuneK(threshold, maxK);
    System.out.println("Testing k = " + k + "...");
    double[] metrics = knn.test(k);
    System.out.println("Precision: " + metrics[0]);
    System.out.println("Recall: " + metrics[1]);
    System.out.println("F1: " + metrics[2]);
```

03

Results



0.55

“Best” F1 Score with sufficient k ~ 638

(Marginal improvement over F1 of around .53 with k ~ 350)

04

Major Obstacles & Key Learnings

Major Obstacles

- **Data Partitioning:**

  Ensuring consistency across sets
- **Optimal Common Words:**

  Determining effective feature sizes
- **Duplicate Entries**

  Unknown duplicates in the raw data set
- **Debugging**

  Problems with implementation

Key Learnings

- **Data Preprocessing:**

  Importance of data preprocessing
- **Model Validation:**

  Intricacies of model validation
- **Complexity of Problem:**

  Function insufficiently complex

Project Planning

Thanks

Do you have any questions?

Resources

- Expanded list of noise/stop words:
  - https://github.com/stopwords-iso/stopwords-en/blob/master/stopwords-en.txt

- Original dataset:
  - https://www.kaggle.com/datasets/surajkarakulath/labelled-corpus-political-bias-hugging-face