# Assignment 3 BUDA 451

Collin Edwards

2025-04-14

## Problem 1: Working with Text Data

In this problem, I work with text. I want to build a model that can say if an Amazon review is good or bad.

### Part I: NLP Basics

**a) What major steps would you take to process text data for sentiment detection?**

I would do these steps: 1. **Clean the text:** Remove punctuation and change all letters to loIrcase to make it uniform. 2. **Tokenize:** Break the text into words because computers work with words, not sentences. 3. **Remove stop words:** Get rid of common words like "the" and "and" that do not add meaning to the text. 4. **Stemming or lemmatization:** Change words to their base form (for example, "running" to "run") to reduce complexity. 5. **Convert text to numbers:** Use methods like TF-IDF to change words into numbers for the computer to use.

### Part II: Term-Frequency Vectors

I have two documents:

- **Document 1:** "This dog food is great, my dog really likes it."
- **Document 2:** "The service was great, but the food was really bad."

First, I remove punctuation and make them loIrcase because computers do not care about punctuation. I also remove stop words like "the" and "is" because they do not help much in understanding the text.

For **Document 1**, the words are:
`this, dog, food, is, great, my, dog, really, likes, it`

Count each word:
- this: 1
- dog: 2
- food: 1
- is: 1
- great: 1
- my: 1
- really: 1
- likes: 1
- it: 1

For **Document 2**, the words are:
`the, service, was, great, but, the, food, was, really, bad`

Count each word:
- the: 2
- service: 1
- was: 2
- great: 1
- but: 1
- food: 1
- really: 1
- bad: 1

These counts are the term-frequency vectors for the documents after removing punctuation and stop words.

### Part III: TF-IDF Transformation

TF-IDF stands for Term Frequency–Inverse Document Frequency. It multiplies the term frequency by the log of (M / dfi), where M is the total number of documents and dfi is the number of documents that have the term i.

**c) What happens if a term appears in only one document?**

If a word is in only one document, then dfi = 1. The log term becomes log(M/1) = log(M). This makes the TF-IDF Iight high, so rare words get more importance in the model. It helps the model focus on unique words that can help decide if a review is good or bad.

**d) What happens if a term appears in every document?**

If a word is in all documents, then dfi = M. The log term becomes log(M/M) = log(1) = 0. The Iight becomes 0. This means common words get very low importance and do not help the model. It helps the model ignore words that do not add much meaning.

**e) What is the purpose of this transformation?**

The goal is to reduce the effect of common words that do not help much in deciding sentiment and give more Iight to rare, important words that can help the model understand the text better. This makes the model more accurate in deciding if a review is good or bad.

---

## Problem 2: Overfitting and Regularization

In this problem, I build models to decide if a movie review is good or bad. I will use three models:
- A Decision Tree
- Logistic Regression
- SVM I will use the same training and test data for all three models. The training data is from Cornell Movie Reviews, which has 10,662 reviews. The test data has 5,331 reviews. I will use TF-IDF to convert the text into numbers.

The training and test data are: - **Training data:** https://raw.githubusercontent.com/binbenliu/Teaching/main/data/cornell_movie/train.csv - **Test data:** https://raw.githubusercontent.com/binbenliu/Teaching/main/data/cornell_movie/test.csv

```
import pandas as pd
# Read the training and test data from the provided URLs.
train_url = "https://raw.githubusercontent.com/binbenliu/Teaching/main/data/cornell_movie/tra
test_url = "https://raw.githubusercontent.com/binbenliu/Teaching/main/data/cornell_movie/test
train_df = pd.read_csv(train_url)
test_df = pd.read_csv(test_url)

# Print the first few rows and the column names.
print("Train Data Head:")
print(train_df.head())
print("Columns in Train Data:")
print(train_df.columns)
```

```
Train Data Head:
   y_label                                               text
0        1  b'as african american detective vergil tibbs q...
1        0  b'the original _babe_ was my favorite movie of...
2        0  b'the king and i , a warner brothers animated ...
3        0  b'susan granger\'s review of " the musketeer "...
4        0  b'in the james bond film " diamonds are foreve...
Columns in Train Data:
Index(['y_label', 'text'], dtype='object')
```

The training data has two columns:
- **y_label**: This tells if the review is positive (1) or negative (0).
- **text**: This contains the review.

First, I load the data. Then I convert the text into numbers using TF-IDF. TF-IDF makes a matrix from the text so I can use it for our models.

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer

# Read the training and test data
train_url = "https://raw.githubusercontent.com/binbenliu/Teaching/main/data/cornell_movie/tra
test_url  = "https://raw.githubusercontent.com/binbenliu/Teaching/main/data/cornell_movie/tes

train_df = pd.read_csv(train_url)
test_df  = pd.read_csv(test_url)

# Print the columns to see what they are.
print("Columns in Train Data:")
print(train_df.columns)

# Our data has two columns: y_label and text.
# I use y_label as the target and text as our input.

# Convert the text data to TF-IDF features.
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(train_df['text'])
X_test = vectorizer.transform(test_df['text'])

# Set y as the label.
y_train = train_df['y_label'].values
y_test  = test_df['y_label'].values

print("Number of training records:", X_train.shape[0])
print("Number of test records:", X_test.shape[0])
```

```
Columns in Train Data:
Index(['y_label', 'text'], dtype='object')
Number of training records: 1400
Number of test records: 600
```

### a) Decision Trees

I set a list of max depth values. For each, I train a decision tree and get the test accuracy.

```
from sklearn import tree
from sklearn.metrics import accuracy_score

max_depths = [2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35, 40, 45, 50]
print("Decision Tree Accuracy for different max_depths:")

for depth in max_depths:
    clf_tree = tree.DecisionTreeClassifier(criterion='entropy', max_depth=depth, random_state
    clf_tree.fit(X_train, y_train)
    y_pred = clf_tree.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    print(f"max_depth = {depth}: Accuracy = {acc:.4f}")
```

```
Decision Tree Accuracy for different max_depths:
max_depth = 2: Accuracy = 0.5817
max_depth = 3: Accuracy = 0.5900
max_depth = 4: Accuracy = 0.6383
max_depth = 5: Accuracy = 0.6317
max_depth = 6: Accuracy = 0.6217
max_depth = 7: Accuracy = 0.6400
max_depth = 8: Accuracy = 0.6450
max_depth = 9: Accuracy = 0.6400
max_depth = 10: Accuracy = 0.6450
max_depth = 15: Accuracy = 0.6417
max_depth = 20: Accuracy = 0.6333
max_depth = 25: Accuracy = 0.6333
max_depth = 30: Accuracy = 0.6333
max_depth = 35: Accuracy = 0.6333
max_depth = 40: Accuracy = 0.6333
max_depth = 45: Accuracy = 0.6333
max_depth = 50: Accuracy = 0.6333
```

```
from sklearn.linear_model import LogisticRegression

reguList = [0.1, 0.5, 1.0, 5, 10, 20, 50, 100]
print("\nLogistic Regression Accuracy for different C values:")

for regu in reguList:
```

```
    clf_lr = LogisticRegression(penalty='l2', C=regu, max_iter=1000, random_state=42)
    clf_lr.fit(X_train, y_train)
    y_pred = clf_lr.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    print(f"C = {regu}: Accuracy = {acc:.4f}")
```

```
Logistic Regression Accuracy for different C values:
C = 0.1: Accuracy = 0.7550
C = 0.5: Accuracy = 0.7967
C = 1.0: Accuracy = 0.8083
C = 5: Accuracy = 0.8417
C = 10: Accuracy = 0.8450
C = 20: Accuracy = 0.8517
C = 50: Accuracy = 0.8517
C = 100: Accuracy = 0.8550
```

```
from sklearn.svm import SVC

print("\nSVM Accuracy for different C values:")

for regu in reguList:
    clf_svm = SVC(C=regu, kernel='rbf', random_state=42)
    clf_svm.fit(X_train, y_train)
    y_pred = clf_svm.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    print(f"C = {regu}: Accuracy = {acc:.4f}")
```

```
SVM Accuracy for different C values:
C = 0.1: Accuracy = 0.4767
C = 0.5: Accuracy = 0.7550
C = 1.0: Accuracy = 0.8150
C = 5: Accuracy = 0.8433
C = 10: Accuracy = 0.8433
C = 20: Accuracy = 0.8433
C = 50: Accuracy = 0.8433
C = 100: Accuracy = 0.8433
```

## Discussion for Problem 2

**Decision Trees:**

- **Key Points:**
  - I tried different `max_depth` values.
  - The best decision tree results Ire reached when the depth was around 8–10.
  - HoIver, the overall accuracy stayed around 64.5%, which is loIr than the other models.
- **Explanation:**
  A decision tree is easy to understand and shows clear rules, but it can miss some important patterns. When the tree is too simple (low depth), it does not capture enough details. When it is too deep, it can overfit the training data. In our tests, even the best tree did not perform very high compared to the others.

---

**Logistic Regression:**

- **Key Points:**
  - I varied the regularization parameter **C** (which controls how simple or complex the model is).
  - The best logistic regression results reached an accuracy of about 85.5% when **C** was high.
- **Explanation:**
  Logistic regression uses a formula that adds up the Iighted features to decide if a review is positive or negative. A larger **C** makes the model more flexible and able to learn from the data. This model did a very good job on our data and had the highest accuracy among the three models.

---

**SVM:**

- **Key Points:**
  - I also varied **C** for the SVM model.

  - The best SVM accuracy was around 84.3% when **C** was 5 or higher.

- **Explanation:**
  The SVM (Support Vector Machine) is a poIrful model that can separate classes Ill. It needed a moderate value of **C** to perform Ill, but its highest accuracy was a bit loIr than that of logistic regression.

---

**Overall Comparison: - Decision Tree:**
- Best accuracy around 64.5%
- Simple and easy to understand but not very accurate. - **Logistic Regression:** - Best accuracy around 85.5%
- Very good at learning patterns and making accurate predictions. - **SVM:** - Best accuracy around 84.3%
- Good at separating classes but not as good as logistic regression. - **Best Model:** - Logistic regression was the best model because it had the highest accuracy. It learned useful patterns from the TF-IDF features of the text data. Decision trees, while simple and easy to interpret, did not perform as well, and SVM also did a good job but did not match the performance of logistic regression.

- **Conclusion:**
  Logistic regression worked the best because it achieved the highest accuracy. It was able to learn useful patterns from the TF-IDF features of the text data. Decision trees, though simple and easy to interpret, did not perform as Ill, and SVM also did a good job but did not match the performance of logistic regression.
- **Future Work:** I would try more complex models like Random Forests or Neural Networks to see if they can do even better. I would also look at different ways to preprocess the text data, like using word embeddings or deep learning methods, to see if they can help improve the model's performance.

## Problem 3: Ensemble Methods

In this problem, we use the Pima Indians Diabetes dataset. The goal is to predict whether a patient has diabetes (1) or not (0) using eight medical features. We will first train a base model—a decision tree—and then build three ensemble methods: Bagging, Random Forest, and AdaBoost. For fair comparisons, we use the same decision tree settings in every model. We report accuracy, precision, recall, and F1 score for each method.

We use a **decision tree** as our base classifier with the following settings: - **Criterion:** entropy
- **Maximum depth:** 4
- **Random state:** 42 (so that results are repeatable)

We will build and compare the following models: - Base Decision Tree (for baseline) - Bagging - Random Forests - AdaBoost

### Step 1: Load and Prepare the Data

```
import pandas as pd
import numpy as np
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Use the raw URLs for the data files
train_url = "https://raw.githubusercontent.com/binbenliu/Teaching/main/IntroAI/data/diabetes_
test_url  = "https://raw.githubusercontent.com/binbenliu/Teaching/main/IntroAI/data/diabetes_

# Load the data
train_df = pd.read_csv(train_url)
test_df  = pd.read_csv(test_url)

# Define the feature columns and target column
x_cols = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
          'BMI', 'DiabetesPedigreeFunction', 'Age']
y_col = 'Outcome'

# Create feature matrices and target vectors
X_train = train_df[x_cols].values
y_train = train_df[y_col].values

X_test = test_df[x_cols].values
y_test = test_df[y_col].values

print("Training data shape:", X_train.shape)
print("Test data shape:", X_test.shape)
```

```
Training data shape: (614, 8)
Test data shape: (154, 8)
```

### Step 2: Train the Base Decision Tree Model

```python
from sklearn.tree import DecisionTreeClassifier

# Train the base decision tree model
clf_tree = DecisionTreeClassifier(criterion='entropy', max_depth=4, random_state=42)
clf_tree.fit(X_train, y_train)
y_pred_tree = clf_tree.predict(X_test)

# Compute performance metrics for the base model
acc_tree = accuracy_score(y_test, y_pred_tree)
prec_tree = precision_score(y_test, y_pred_tree)
rec_tree = recall_score(y_test, y_pred_tree)
f1_tree = f1_score(y_test, y_pred_tree)

print("Base Decision Tree Performance:")
print(f"Accuracy:  {acc_tree:.4f}")
print(f"Precision: {prec_tree:.4f}")
print(f"Recall:    {rec_tree:.4f}")
print(f"F1 Score:  {f1_tree:.4f}")
```

```
Base Decision Tree Performance:
Accuracy:  0.7403
Precision: 0.6667
Recall:    0.5455
F1 Score:  0.6000
```

### Step 3: Train the Bagging Model Ensemble

Bagging (Bootstrap Aggregation) builds many copies of the decision tree model on different random samples from the training data. The final prediction is made by averaging the predictions from all trees. This reduces errors and increases stability.

```python
from sklearn.ensemble import BaggingClassifier

clf_bag = BaggingClassifier(
    estimator=DecisionTreeClassifier(criterion='entropy', max_depth=4, random_state=42),
    n_estimators=500,  # number of trees
    max_samples=100,   # number of samples for each tree
    bootstrap=True,    # sample with replacement
    random_state=42
)
```

```
clf_bag.fit(X_train, y_train)
y_pred_bag = clf_bag.predict(X_test)

# Compute performance metrics for the Bagging model
acc_bag = accuracy_score(y_test, y_pred_bag)
prec_bag = precision_score(y_test, y_pred_bag)
rec_bag = recall_score(y_test, y_pred_bag)
f1_bag = f1_score(y_test, y_pred_bag)

print("Bagging Performance:")
print(f"Accuracy:  {acc_bag:.4f}")
print(f"Precision: {prec_bag:.4f}")
print(f"Recall:    {rec_bag:.4f}")
print(f"F1 Score:  {f1_bag:.4f}")
```

```
Bagging Performance:
Accuracy:  0.7597
Precision: 0.7250
Recall:    0.5273
F1 Score:  0.6105
```

### Step 4: Train the Random Forest Model Ensemble

Random Forests build many decision trees and average their predictions. It is a more advanced version of bagging that uses random subsets of features for each tree, which helps reduce overfitting.

```
from sklearn.ensemble import RandomForestClassifier

clf_rf = RandomForestClassifier(
    criterion='entropy', max_depth=4, n_estimators=500, random_state=42
)
clf_rf.fit(X_train, y_train)
y_pred_rf = clf_rf.predict(X_test)

# Compute performance metrics for the Random Forest model
acc_rf = accuracy_score(y_test, y_pred_rf)
prec_rf = precision_score(y_test, y_pred_rf)
rec_rf = recall_score(y_test, y_pred_rf)
f1_rf = f1_score(y_test, y_pred_rf)
```

```
print("Random Forest Performance:")
print(f"Accuracy:  {acc_rf:.4f}")
print(f"Precision: {prec_rf:.4f}")
print(f"Recall:    {rec_rf:.4f}")
print(f"F1 Score:  {f1_rf:.4f}")
```

```
Random Forest Performance:
Accuracy:  0.7403
Precision: 0.7027
Recall:    0.4727
F1 Score:  0.5652
```

**Step 5: Train the AdaBoost Model Ensemble**

```
from sklearn.ensemble import AdaBoostClassifier

clf_adaboost = AdaBoostClassifier(
    estimator=DecisionTreeClassifier(criterion='entropy', max_depth=4, random_state=42),
    n_estimators=500,
    random_state=42
)

clf_adaboost.fit(X_train, y_train)
y_pred_adaboost = clf_adaboost.predict(X_test)

# Compute performance metrics for AdaBoost
acc_adaboost = accuracy_score(y_test, y_pred_adaboost)
prec_adaboost = precision_score(y_test, y_pred_adaboost)
rec_adaboost = recall_score(y_test, y_pred_adaboost)
f1_adaboost = f1_score(y_test, y_pred_adaboost)

print("AdaBoost Performance:")
print(f"Accuracy:  {acc_adaboost:.4f}")
print(f"Precision: {prec_adaboost:.4f}")
print(f"Recall:    {rec_adaboost:.4f}")
print(f"F1 Score:  {f1_adaboost:.4f}")
```

```
AdaBoost Performance:
Accuracy:  0.7403
```

```
Precision: 0.6744
Recall:    0.5273
F1 Score:  0.5918
```

## Overall Discussion Conclusion of Problem 3

**Base Decision Tree:**
- Used a decision tree with `max_depth=4`.
- Achieved an accuracy of about **74.03%** with 66.67% precision, 54.55% recall, and an F1 score of 60.00%.
- This serves as our baseline model.

**Bagging:**
- Builds many trees on different random samples and averages their predictions.
- Improved accuracy to **75.97%** and precision to 72.50%.
- It helps reduce errors by combining multiple models.

**Random Forest:**
- Similar to bagging but randomly selects features at each split.
- Accuracy remained around **74.03%** with slightly higher precision (70.27%) but lower recall (47.27%).
- It did not show significant improvement over the base model in this experiment.

**AdaBoost:**
- Builds a series of weak classifiers that focus on previous mistakes.
- Performance stayed similar to the base tree with an accuracy of **74.03%** and comparable precision, recall, and F1 scores.

**Overall Conclusion:**
Ensemble methods like Bagging provided a small improvement over the base decision tree, while Random Forest and AdaBoost performed similarly to the base model in this case. In our experiment, Bagging appeared to have the best overall performance.