NaiveBayes

November 10, 2024

1 Probability Theory and the Naive Bayes algorithm

In this assignment, you will:

1. Solve a probability theory question.
2. Implement the Naive Bayes algorithm for text classification, where probilities are estimated using Laplace smoothing.
3. Evaluate the Naive Bayes model on two tasks: spam filtering and sentiment analysis.
4. Bonus points.
5. Analyze the results.

1.1 Write Your Name Here:

2 Submission instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of ll cells).
4. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
5. Once you've rerun everything, select File -> Download as -> PDF via LaTeX and download a PDF version showing the code and the output of all cells, and save it in the same folder that contains the notebook file.
6. Look at the PDF file and make sure all your solutions are there, displayed correctly.
7. Submit **both** your PDF and the notebook file .ipynb on Canvas.
8. Make sure your your Canvas submission contains the correct files by downloading it after posting it on Canvas.

2.1 Probability Theory (100 points)

Given that: - 10 out of every 1,000 women at age forty who participate in routine screen have breast cancer.

- 8 of every 10 women with breast cancer will get a positive mammography.
- 95 out of every 990 women without breast cancer will also get a positive mammography.

Answer the following question: - What is the likelihood that a forty year old woman has breast cancer, given that she got a positive mammography in a routine screening?

2.1.1 Solution

Your solution goes here. Show your work in detail. Take advantage of the Jupyter Notebook markdown language, which can also process Latex and HTML, to format your math so that it looks professional.

2.2 The Naive Bayes algorithm (50 + 50 points)

Given a set of training examples `X` and their labels `y`, compute the class `priors` and class-conditinal word `likelihoods`. For example, `priors[k]` should contain the probability $p(C_k)$ for class k, whereas `likelihoods[k, j]` should contained the probability $p(w_j|C_k)$ that the word with vocabulary index j appears in a document that has class C_k. For computing word likelihoods, use Laplace smoothing as discussed in class.

[ ]:
```
import numpy as np
```

[1]:
```
def nb_train(X, y, K, V):
    """Naive Bayes training function.
    Args:
        X: The list of training example, where each example is a dictionary␣
 ↪mapping features (word indeces) to counts.
        y: The list of labels.
        K: The number of classes.
        V: The number of features (size of vocabulary).

    Returns:
        priors (np.ndarray): A vector of class priors, one prior per class.
        likelihoods (np.ndarray): A 2D array of class-conditional word␣
 ↪probabilities, one row for each class.

    """
    priors = np.zeros(K)
    likelihoods = np.zeros((K, V))

    # YOUR CODE HERE




    return priors, likelihods
```

Implement the Naive Bayes prediction function that takes as input the Naive Bayes parameters (priors and likelihoods) and returns a list with the labels (0 foe negative and 1 for positive) predicted for the test examples in input data X.

[3]:
```
def nb_test(model, X):
    """Naive Bayes prediction function.
    Args:
```

```
        model: The tuple (log_priors, log_likelihoods) containing the NB model␣
↪parameters (log probabilities).
        X: The list containing all test examples, where each example is a␣
↪dictionary mapping the word id to its corresponding count.

    Returns:
        pred: The list of predicted labels.
    """
    # YOUR CODE HERE
    pred = [None for x in X]



    return pred
```

2.3 Saving NB model parameters in a text file (20 points)

[ ]:
```
def save_model(model, fmodel):
    """
    Args:
        model: The tuple (priors, likelihoods) containing the NB model␣
↪parameters (probabilities).
        fmodel: The name of the text file where the log probabilities will be␣
↪stored.

        The parameters should be stored as follows:
        - The first line in the file contains the number of classes K and the␣
↪number of features V:
            K V
        - The second line contains the log of the prior class probabilities␣
↪separated by spaces:
            log(p(C_1)) log(p(C_2) ... log(p(C_K))
        - The remaining lines contain the log of the class conditional␣
↪probabilities, one line for each class, starting with the class id:
            C_1 log(p(w_1|C_1)) log(p(w_2|C_1)) ... log(p(w_V|C_1))
            C_2 log(p(w_1|C_2)) log(p(w_2|C_2)) ... log(p(w_V|C_2))
            ...
            C_K log(p(w_1|C_K)) log(p(w_2|C_K)) ... log(p(w_V|C_K))
    """
    # YOUR CODE HERE
```

2.4 Reading NB model parameters from a text file (20 points)

[4]:
```
def read_model(fmodel):
    """
    Args:
        fmodel: The name of the file storing the NB model parameters.

    Return
        log_priors, log_likelihoods: The NB model parameters (log version).
    """

    log_priors, log_likelihoods = None, None
    with open(fmodel, "r", encoding="utf-8") as fi:
        line = fi.readline().split()
        K, V = int(line[0]), int(line[1])
        log_priors = np.ones(K)
        log_likelihoods = np.ones((K, V))

        # YOUR CODE HERE









    return log_priors, log_likelihoods
```

2.5 Creating and reading the vocabulary (30 + 20 points)

Implement a function `create_vocabulary()` that takes as input the path to the training file and the path to a file that will store the vocabulary created by the function. The vocabulary file should contain a list of all the (pre-processed) tokens that appear in the training examples at least `mincount` times, together with their counts. The file should contain one token per line in the format <id> <token>, where each token is associated a unique integer identifier. The tokens should be listed in increasing order of their identifiers, starting from 0.

[ ]:
```
def create_vocabulary(finput, foutput, mincount = 0):
    """Function that creates the vocabulary and saves it into a file.
    Args:
        finput (string): The path to the training file.
        foutput (string): The path to the output file where the vocabulary will␣
 ↪be saved
        mincount (int): The minimum count number.
    """

    # Initialize the vocabulary to an empty dictionary, create it from the␣
 ↪input file.
```

```
    # The dictionary will map each token to its count.
    vocab = {}
    with open(finput, "r", encoding="utf-8") as fi:
        for line in fi:
            # YOUR CODE HERE


    # Write the vocabulary into the output file, one token per line in the␣
↪format <id> <token>
    #   where each token is associated a unique integer identifier, starting␣
↪from 0.
    #   Only include tokens that appear at least mincount times in training␣
↪examples.
    with open(foutput, "w", encoding="utf-8") as fo:
        # YOUR CODE HERE
```

Implement a function `load_vocabulary()` the reads the vocabulary from a file into a dictionary that maps each token to its id.

[ ]:
```
def load_vocabulary(finput):
    """Function that reads the vocabulary from an input file.
    Args:
        finput (string): The path to the vocabulary file.

    Returns:
        vocab: A Python dictionary containing the vocabulary, mapping string␣
↪tokens to their id's.
    """

    vocab = {}
    with open(finput, "r", encoding="utf-8") as fi:
        for line in fi:
            # YOUR CODE HERE




    return vocab
```

2.6 Creating and reading examples using the SVM sparse format (30 + 20 points)

Implement a function `create_svm_format()` that reads an input file (e.g. training or testing data) and for each example in the file it creates a sparse feature vector representation wherein each example is represented as one line in the file using the format <label> <id1>:<val1> <id2>:<val2> ..., where the *id*'s are listed in increasing order and correspond only to vocabulary tokens that appear

in that example. The values *val* correspond to the number of times the token *id* appeared in that example. The <label> is 0 for negative examples and 1 for positive examples.

[ ]:
```
def create_svm_format(finput, foutput, vocab):
    """Function that creates the sparse feature vector representation of a␣
 ↪dataset.
    Args:
        finput (string): The path to the file containing the dataset (e.g.,␣
 ↪training or test).
        foutput (string): The path to the output file where the new format of␣
 ↪the dataset will be saved.
        vocab (dict): The Python dictionary containing the vocabulary.
    """

    with open(finput, "r", encoding="utf-8") as fi, open(foutput, "w",␣
 ↪encoding="utf-8") as fo:
        for line in fi:
            # YOUR CODE HERE
```

The function `read_examples()` below reads all examples from a file with sparse feature vectors and returns a tuple `(data, labels)`. The `data` is a list containing all examples, where each example is a dictionary mapping the feature (word) *id* to its corresponding value (count). The `labels` is a list containing the corresponding labels. You can use this function to verify that your implementation of `create_svm_format()` is correct.

[2]:
```
def read_examples(file_name):
    """Function that read the sparse feature vector representation of a dataset.
    Args:
        file_name (string): the path to the file containing a set of examples␣
 ↪in the sparse feature vector format.
        nfeatures (int): the total number of features.

    Returns:
        X: a list containing all examples, where each example is a dictionary␣
 ↪mapping the word id to its corresponding count.
        t: a list containing the corresponding labels.
    """

    X = []
    t = []
    with open(file_name, "r", encoding="utf-8") as fi:
        for line in fi:
```

```
            label, *features = line.strip().split(" ")
            t.append(int(label))
            fdict = {}
            for x in features:
                index, value = x.split(":")
                fdict[int(index)] = int(value)
            X.append(fdict)

    return X, t
```

2.7 Spam Filtering with Naive Bayes

In this problem, you will train and evaluate spam classifiers using the Naive Bayes algorithm. The dataset contains two files: *spam_train.txt* with 4,000 training examples and *spam_test.txt* with 1,000 test examples. The dataset is based on a subset of the Spam Assassin Public Corpus. Each line in the training and test files contains the pre-processed version of one email. The line starts with the label, followed by the email tokens separated by spaces.

Figure 1 shows a sample source email, while Figure 2 shows its pre-processed version in which web addresses are replaced with the "httpaddr" token, numbers are replaced with a "number" token, dollar amounts are replaced with "dollarnumb", and emaill addresses are replaced with "emailaddr". Furthermore, all words are lower-cased, HTML tags are removed, and words are reduced to their stems i.e. "expecting", "expected", "expectation" are all replaced with "expect". Non-words and punctuation symbols are removed.

[ ]:
```
# Create vocabulary from training examples, use only tokens appearing at least␣
 ↪30 times in the training data.
create_vocabulary("../data/spam/spam_train.txt", "../data/spam/spam_vocab.txt",␣
 ↪30)

# Read vocabulary.
vocab = load_vocabulary("../data/spam/spam_vocab.txt")
print("Number of features is:", len(vocab))

# Generate sparse format file.
create_svm_format("../data/spam/spam_train.txt", "../data/spam/spam_train_svm.
 ↪txt", vocab)
create_svm_format("../data/spam/spam_test.txt", "../data/spam/spam_test_svm.
 ↪txt", vocab)

# Read the training and test examples from the sparse format files.
Xtrain, ytrain = read_examples("../data/spam/spam_train_svm.txt")
Xtest, ytest = read_examples("../data/spam/spam_test_svm.txt")


################## Naive Bayes experiment ##################

# Train the Naive Bayes model and save its parameters.
```

```
print("** Naive Bayes **")
nb_model = nb_train(Xtrain, ytrain, 2, len(vocab))
save_model(nb_model, "spam_model.txt")

# Load the Naive Bayes model and evaluate its accuracy on the test examples.
nb_model = read_model("spam_model.txt")
pred = nb_test(nb_model, Xtest)

acc = # YOUR CODE HERE

print("Spam filtering accuracy is: %.4f\n" % acc)
```

2.8 Sentiment Analysis with Naive Bayes (50 bonus points)

In this problem, you will train and evaluate sentiment classifiers using the Naive Bayes algorithm. The dataset contains two files: *imdb_sentiment_train* with 1,500 training examples and *imdb_sentiment_test* with 1,500 test examples. Each line in the training and test files contains one example that starts with the label, followed by the text of the movie review.

It is important that the moview review is first pre-processed as follows:

1. Tokenize the text using text tokenizer included with libraries such as Spacy or sklearn.
2. Each token is lowercased.
3. Only use tokens that contain only letters (ignore tokens that contain numbers punctuation symbols, and other symbols).

2.9 Spam Filtering with Naive Bayes using `sklearn` (30 bonus points)

Train and test the sklearn NaiveBayes algorithm on the same *Spam vs. Non-spam* classification problem described above.

The following functionality from the `sklearn.svm` will be useful:

- `sklearn.naive_bayes.MultinomialNB`: This is the main class used for the NB classification models. Make sure that you properly map the NB hyper-parameters to the parameters in the constructor of this class.
- `fit()`: This is the function used to train the classifier.
- `predict(x)`: This is used to calculate the (binary) label for sample x.

[ ]:
```
# YOUR CODE HERE
```

2.10 Analysis of results (20 points)

Include here a nicely formatted report of the results, comparisons. Include explanations and any insights you can derive from the algorithm behavior and the results. This section is important, so make sure you address it appropriately.

Take advantage of the Jupyter Notebook markdown language, which can also process Latex and HTML, to format your report so that it looks professional.

2.11 Anything extra goes here

For example: - evaluating Naive Bayes by also using bigrams, i.e. combinations of two words, as features.

- evaluating Naive Bayes with different values for the mincount threshold and reporting the value that obtains the best performance.
- evaluating NB when ignorign stop words included in the file `stop_words.txt`.

[ ]:

[ ]: