```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// NAME: Jared Joyal
// EMAIL: jared.joyal@comcast.net
// © 2019 DigiPen, All Rights Reserved.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// alpha and beta are smoothing parameters
static const unsigned alpha = 1;
static const unsigned beta = 2;
// threshold is the lowest probability at which we will mark an email as spam
static const float threshold = 0.8f;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// DESCRIPTION: removes special characters, leaving only whitespace and alpha characters
// subjects: vector of subjects in which to remove special characters
static void RemoveNonAlpha(std::vector<std::string> &subjects)
{
  // function that returns true if character is not alpha nor whitespace
  static const std::function<bool(char)> notAlphaNorSpace =
    [](unsigned char c) { return !std::isalpha(c) && !std::isspace(c); };

  // remove every character from every subject that is not alpha (or whitespace)
  for (auto &sub : subjects)
    sub.erase(std::remove_if(sub.begin(), sub.end(), isAlphaOrSpace), sub.end());
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// DESCRIPTION: sets all alpha characters from A-Z to a-z
// subjects: vector of subjects in which to set to lowercase
static void StringsToLower(std::vector<std::string> &subjects)
{
  // for every character of every subject, set character to lower
  for (auto &sub : subjects)
    std::transform(sub.begin(), sub.end(), sub.begin(), tolower);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// DESCRIPTION: if a word does not appear in a dictionary file, it is removed from the list of words we consider
// filename: name of dictionary file
// frequencies: list of words with frequencies of appearing in spam/ham emails
static void TrimWords(const std::string &filename, std::map<std::string, std::pair<unsigned, unsigned>> &frequencies)
{
  // load dictionary of words into ordered set for quick lookup
  std::set<std::string> dictionary;
  std::ifstream dictFile(filename);
  if (dictFile.is_open())
  {
    std::string line;
    while (std::getline(dictFile, line))
      dictionary.insert(line);
  }

  // get rid of words that are not found in dictionary
  auto it = frequencies.begin();
  while (it != frequencies.end())
  {
    if (dictionary.find(it->first) == dictionary.end())
      it = frequencies.erase(it);
    else
      ++it;
  }
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// DESCRIPTION: runs spam filter on a vector of uncategorized email subjects, determining whether they are spam or ham
// words: list of words to consider for spam filter
// subjects: email subjects to label as either spam or ham
// yVec, y0, zVec, z0: pre-computed values per the algorithm
// spamProbability, hamProbability: pre-computed P(spam) and P(ham), respectively
static unsigned RunFilter(const std::vector<std::string> &words, std::vector<std::string> subjects,
                          const std::vector<float> &yVec, float y0, const std::vector<float> &zVec, float z0,
                          float spamProbability, float hamProbability)
{
  unsigned markedAsSpam = 0;

  // for every subject line
  for (const auto &subject : subjects)
  {
    // calculate 'a' vector by seeing if words are in subject line
    std::vector<unsigned> aVec;
    for (const auto &word : words)
      aVec.push_back(subject.find(word) != std::string::npos ? 1 : 0);

    // calculate P(spam|X=aVec) using pre-calculated condensed equation
    // (DotProduct helper function not shown)
    float expAdotY = std::expf(DotProduct(aVec, yVec) + y0);
    float probOfSpam = expAdotY * spamProbability /
      (expAdotY * spamProbability + std::expf(DotProduct(aVec, zVec) + z0) * hamProbability);

    // check against threshold
    if (probOfSpam >= threshold)
      ++markedAsSpam;
  }

  return markedAsSpam;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// DESCRIPTION: this program encapsulates the full process of parsing subjects out of emails, filtering the data by
//              omitting special characters and non-real words, training an algorithm based on known data, and testing
//              it using a separate set of known data (not used for training)
int main()
{
  // use std::filesystem to parse multiple files in one directory, creating vector of subject lines
  // every fourth subject is placed in a testing set instead to test the learned algorithm
  // (helper function not shown)
  std::vector<std::string> spamTraining, spamTesting, hamTraining, hamTesting;
  ParseEmails("../spam", spamTraining, spamTesting);
  ParseEmails("../ham", hamTraining, hamTesting);

  // remove all characters that are not letters (or spaces)
  RemoveNonAlpha(spamTraining);
  RemoveNonAlpha(spamTesting);
  RemoveNonAlpha(hamTraining);
  RemoveNonAlpha(hamTesting);

  // set all subjects to lowercase
  StringsToLower(spamTraining);
  StringsToLower(spamTesting);
  StringsToLower(hamTraining);
  StringsToLower(hamTesting);

  // tokenize words from subjects, and tally how often they show up
  // map has keys of words, and values are pairs of (frequency in spam email, frequency in ham email)
  // (helper function not shown)
  std::map<std::string, std::pair<unsigned, unsigned>> spamAndHamFrequencies;
  ParseWords(spamTraining, spamAndHamFrequencies, true);
  ParseWords(hamTraining, spamAndHamFrequencies, false);
```

```
  // trim words that are not real
  TrimWords("../dictionary.txt", spamAndHamFrequencies);

  // put the keys in a vector, for index lookup convenience
  std::vector<std::string> words;
  std::transform(spamAndHamFrequencies.begin(), spamAndHamFrequencies.end(), std::back_inserter(words),
    [](std::pair<std::string, std::pair<unsigned, unsigned>> pair) { return pair.first; });

  // calculate probabilities of words given spam/ham
  // also calculate probabilities of spam/ham given word
  std::vector<float> wordGivenSpamProbabilities, wordGivenHamProbabilities,
                     spamGivenWordProbabilities, hamGivenWordProbabilities;
  unsigned totalSpam = spamTraining.size();
  unsigned totalHam = hamTraining.size();
  const float spamProbability = totalSpam / float(totalSpam + totalHam);
  const float hamProbability = totalHam / float(totalSpam + totalHam);
  for (const auto &freq : spamAndHamFrequencies)
  {
    wordGivenSpamProbabilities.push_back((freq.second.first + alpha) / float(totalSpam + beta));
    wordGivenHamProbabilities.push_back((freq.second.second + alpha) / float(totalHam + beta));
    spamGivenWordProbabilities.push_back(wordGivenSpamProbabilities.back() * spamProbability /
      (wordGivenSpamProbabilities.back() * spamProbability + wordGivenHamProbabilities.back() * hamProbability));
    hamGivenWordProbabilities.push_back(wordGivenHamProbabilities.back() * hamProbability /
      (wordGivenSpamProbabilities.back() * spamProbability + wordGivenHamProbabilities.back() * hamProbability));
  }

  // do some pre-calculations, namely the probabilities of each word given spam/ham
  // this allows us to simplify the Naive Bayes formula, using dot product and logarithmic calculations instead of
  //   hundreds of multiplications, additions, and divisions per email to test
  std::vector<float> yVec, zVec;
  float y0 = 0.0f, z0 = 0.0f;
  for (unsigned i = 0; i < wordGivenSpamProbabilities.size() /* same size as wordGivenHamProbabilities */; ++i)
  {
    y0 += std::logf(1.0f - wordGivenSpamProbabilities[i]);
    z0 += std::logf(1.0f - wordGivenHamProbabilities[i]);
    yVec.push_back(std::logf(wordGivenSpamProbabilities[i] / (1.0f - wordGivenSpamProbabilities[i])));
    zVec.push_back(std::logf(wordGivenHamProbabilities[i] / (1.0f - wordGivenHamProbabilities[i])));
  }

  // test filter against testing data
  unsigned spamMarkedAsSpam = RunFilter(words, spamTesting, yVec, y0, zVec, z0, spamProbability, hamProbability);
  unsigned hamMarkedAsSpam = RunFilter(words, hamTesting, yVec, y0, zVec, z0, spamProbability, hamProbability);

  // output statistical results of testing
  totalSpam = spamTesting.size();
  totalHam = hamTesting.size();
  std::cout << "Spam Marked As Spam: " << spamMarkedAsSpam << std::endl;
  std::cout << "Ham Marked As Spam: " << hamMarkedAsSpam << std::endl;
  std::cout << "Total Spam: " << totalSpam << ", Total Ham: " << totalHam << std::endl;
  float accuracy = (totalHam - hamMarkedAsSpam + spamMarkedAsSpam) / float(totalSpam + totalHam);
  float precision = spamMarkedAsSpam / float(spamMarkedAsSpam + hamMarkedAsSpam);
  float recall = spamMarkedAsSpam / float(totalSpam);
  std::cout << "Accuracy: " << accuracy << std::endl;
  std::cout << "Precision: " << precision << std::endl;
  std::cout << "Recall: " << recall << std::endl;

  return 0;
}
```