Spam detector:

The above code uses the Pandas library to import a CSV file ("C:/Users/dridi/Downloads/Youtube01-Psy.csv") and stores the data in a DataFrame named d. Then it displays the last 10 rows of the DataFrame using the tail(10) method.

The rest of the code calculates the number of positive and negative comments in the d DataFrame. To do this, it uses the len() function to calculate the number of rows where the CLASS column value is equal to 1 or 0. Thus, len(d.query('CLASS == 1')) calculates the number of positive comments (where the value of CLASS is equal to 1) and returns 175. Similarly, len(d.query('CLASS == 0')) calculates the number of negative comments (where the value of CLASS is equal to 0) and also returns 175. Finally, len(d) calculates the total number of comments in the DataFrame d, which is equal to 350.

Next, the code uses the scikit-learn library to create a word occurrence matrix using the CountVectorizer() class. The fit_transform() method is used to apply the CountVectorizer() to the comment text stored in the CONTENT column of d and store the result in the dvec variable. The matrix dvec is of size 350x1418, which means that there are 350 comments in the DataFrame d and 1418 unique words in those comments. The matrix is stored as a hollow matrix to save memory.

Next, the code uses the build_analyzer() method of CountVectorizer() to create a function that turns a string into a list of words. Next, it displays the last comment in the d DataFrame (stored in line 349 of d['CONTENT']) using print(d['CONTENT'][349]), then applies the analyze function to that comment to return the list of words it contains.

The rest of the code starts by creating an instance of the scikit-learn library's CountVectorizer() class to create a vector representation of comments. Next, a small corpus of words is created as an X_train list containing frequently used words in YouTube comments.

The fit() method is used to train the vectorizer on the data in the X_train list, which builds a vocabulary of all the unique words in X_train. The get_feature_names() method is used to get a list of all words in the created vocabulary. Then the list of unique words is displayed using the print() function.

The rest of the code shuffles the data in the DataFrame d using the sample() method to select a random sample of rows. The data is then split into a training set (d_train) and a test set (d_test). The comments of the training set are represented as vectors using the vectorizer previously created with the fit_transform() method. The test set comments are also represented as vectors using the transform() method.

Next, the code uses the RandomForestClassifier machine learning algorithm from the scikit-learn library to train a model on the training data. The fit() method is used to train the model on the training data d_train_att with the corresponding labels d_train_label. The number of trees in the RandomForest classifier is set to 80 using the n_estimators parameter.

Subsequently, the code uses the score() method to evaluate the accuracy of the trained model on the test data d_test_att and the corresponding labels d_test_label. The accuracy is calculated as the ratio of the model's correct predictions to the total number of examples in the test set. The resulting accuracy is 0.96, which means that the model correctly predicted the class of 96% of the comments in the test set.

From line 18 the necessary libraries are imported, including scikit-learn, pandas and numpy. Next, the data is loaded from five different csv files and combined into a single dataset using the concat()function of the pandas library. The resulting dataset has 1956 rows, with 1005 rows in class 1 (spam) and 951 rows in class 0 (no spam).

The next section of code uses the scikit-learn confusion_matrix()function to compute the confusion matrix for the predicted and actual labels in the test set. The confusion matrix shows the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) of the classifier predictions. In this example, the output shows that there were 26 true positives and 22 true negatives, while there were no false positives or false negatives.

Next the section uses scikit-learn's cross_val_score()function to perform cross-validation on the data set. Cross-validation is a technique for evaluating the performance of a model by dividing the data into training and test sets multiple times. In this example, the code uses 5-fold cross-validation (cv=5) and prints the average accuracy score and 95% confidence interval for the model. The output shows an accuracy of 0.96 with a standard deviation of 0.07.

Next, the data is shuffled and split into d_content(the comments) and d_label(the corresponding labels).

The next section sets up a pipeline for processing the data. A pipeline is a way to organize multiple steps in a machine learning workflow. In this example, the pipeline consists of two steps: first, the CountVectorizer()function is used to transform the comments into a matrix of token counts, and then the RandomForestClassifier()function is used to classify the comments into spam or non-spam. The make_pipeline()function can also be used to configure the pipeline.

From line 31: The first line of code predicts the category of a new input text "please subscribe to my channel" and returns a single prediction of 1, suggesting that the text belongs to the positive category.

The following code block uses cross-validation to evaluate the performance of the model on a larger dataset. The model achieves an average accuracy of 0.96 with a standard deviation of 0.01, indicating that it performs well on the data set.

The third block of code adds a TfidfTransformer to the pipeline to improve the performance of the model. TfidfTransformer calculates the inverse document frequency-frequency term (TF-IDF) values for the features, which can help identify the most important features for the classification task. The cross-validation accuracy of the model improves slightly to 0.96 with a standard deviation of 0.03.

The fourth code block performs a grid search to find the best hyperparameters for the model. The grid search tests various combinations of hyperparameters, such as maximum number of features, n-gram range, empty words, use of IDF, and number of estimators in the random forest classifier. The best hyperparameters found are: max_features = 2000, ngram_range = (1, 1), stop_words = 'english', n_estimators = 100, use_idf = False. The model achieves an accuracy of 0.96 with these hyperparameters, which is identical to the accuracy of the previous cross validation.

Overall, the code trains a machine learning model to classify text data, evaluates its performance using cross-validation, improves its performance using a TfidfTransformer, and optimizes its hyperparameters using a grid search. The final model achieves high accuracy on the dataset, suggesting that it can generalize well to new, unseen data.

Sentiment Analysis:

The code begins by importing the necessary libraries to perform text processing and machine learning tasks. Then, it configures the logging format to display logging information at a certain severity level. Next, it reads a text data file (yelp_labelled.txt) line by line using a for loop, using the enumerate() method to get the element number (item_no) and the corresponding line (line).

The loop then uses the print() method to display the element number and the corresponding line. The code then creates an empty list named 'sentences' which will be used to store preprocessed text data for machine learning.

Next, the code loops through three text data files (yelp_labelled.txt, amazon_cells_labelled.txt, imdb_labelled.txt) using another nested for loop. It reads each line of each file, uses the strip() method to remove unnecessary whitespace at the beginning and end of the line, and splits each line into two parts (the text string and the sentiment label) using the split('\t') method.

The code then converts the text string to lowercase, removes apostrophes and special characters using regular expressions, and splits the text string into a list of words (tokens) using the split() method. Next, the code creates a TaggedDocument object from the list of words and sentiment label (in the format "%s_%d" % (fname, item_no)) and adds this object to the 'sentences' list. Finally, the code also adds the sentiment label (0 or 1) to another list named 'sentiments'.

Continuing the analysis, the previous code creates a PermuteSentences object that inherits from the Python object class and defines a iter method. This method allows iterating over the elements of 'sentences' in a random order using the shuffle method of the random module.

Next, a Doc2Vec model is created by taking 'permuter' as the corpus and specifying that the minimum number of occurrences required for a word to be added to the vocabulary is 1 (with the argument min_count=1). The model will therefore consider all words present in the corpus.

The line logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO) configures Python's logging module to display information messages (INFO) with a given format. The information messages that follow indicate that the model is collecting the words and their frequencies in the corpus, creating a vocabulary, and preparing data for model training.

Note that the path to the "yelp_labelled.txt" file is specified in line 6, and the code uses the open() function to read the contents of this file. Finally, the Doc2Vec model will be used to encode sentences and enable tasks such as sentence similarity search or sentence classification.

Finally, the code model.wv.most_similar('tasty') uses the Word2Vec model trained on a data corpus to find the words most similar to 'tasty'. The result is a list of 10 words with their similarity score, sorted in descending order of score. The words most like 'tasty' according to this model are:

• 'there' with a similarity score of 0.9985021948814392

• 'show' with a similarity score of 0.9984488487243652

Etc….

This suggests that these words tend to appear in the same context as the word 'tasty' in the training corpus used to train the Word2Vec model.

Sentiment Analysis New

The provided code is an example of using the Python library "gensim" for working with natural language processing models, specifically the Word2Vec models.

Firstly, the first line sets up a logging system to record informational messages (logging.INFO level) in a standard format that includes the date and time, severity level, and the message itself.

Next, the code begins by setting up a logging system using the logging module. The following line then loads the Word2Vec model from the binary file "GoogleNews-vectors-negative300.bin" using Gensim's load_word2vec_format() method. This model has a vocabulary size of 3 million words, with each word represented by a 300-dimensional vector.

The three following commands extract word vectors for "cat", "dog", and "spatula" using key indexing (gmodel['cat'], gmodel['dog'], and gmodel['spatula']). Each vector is a NumPy array of 300 elements representing the vector representation of each word.

The next line uses Gensim's similarity() method to calculate the cosine similarity between "cat" and "dog". Cosine similarity measures the similarity between two vectors by using the angle between them. The closer the cosine similarity value is to 1, the more similar the two vectors are. In this case, the cosine similarity between "cat" and "dog" is 0.7609457, indicating that the two words are quite similar.

The last line of the code also uses Gensim's similarity() method to calculate the cosine similarity between "cat" and "spatula". In this case, the cosine similarity is much lower, at only 0.12412614. This indicates that the two words are very different from each other in terms of meaning.

Then the code defines a function extract_words() that is used to clean and preprocess raw text sentences. Next, the code loads unsupervised training data from several sources and stores each sentence in a data structure called "TaggedDocument", which contains a list of words and a label for each document. This unsupervised data can then be used to train a Doc2Vec model, which is a natural language processing algorithm for creating vector representations of entire sentences and documents. Finally, the last line of code prints the total number of documents in the unsup_sentences list and the first ten documents.

At the line 11 we define a function **extract_words(sent)** that preprocesses a sentence by converting it to lowercase, removing HTML tags and punctuation, and splitting it into words.

The code then creates an empty list called **unsup_sentences** to hold the training data. It loops over three different directories containing movie reviews from the sources: the IMDB dataset, the Cornell movie review dataset, and the Rotten Tomatoes dataset. It reads each file in each directory that ends with **.txt**, preprocesses the text using **extract_words**, and adds it to the **unsup_sentences** list as a **TaggedDocument** object.

A **TaggedDocument** object consists of a list of words and a tag. The tag is a unique identifier that can be used to retrieve the document later. In this case, the tag consists of the directory path and the filename.

Finally, the total number of **TaggedDocument** objects in **unsup_sentences** is printed, which is **175325**.

The code begins by using different sources to construct a set of unlabeled sentences. These sources include IMDB data, Cornell movie review data, and Rotten Tomatoes data.

Next, the code uses the trained model to calculate cosine similarities between pairs of sentences. Specifically, it calculates the similarity between two pairs of sentences: the first is "This place is not worth your time, let alone Vegas." and "Service sucks.", while the second is "Highly recommended." and "Service sucks.".

The cosine similarity results are 0.48211202 and 0.28899333, respectively. These values indicate that the first pair of sentences is closer than the second pair of sentences.

The code continues by reading files containing labeled sentences for sentiment classification. The sentences and labels are stored in the "sentences" and "sentiments" lists. The sentences are then transformed into vectors using the trained Doc2Vec model, which is used for sentiment classification.

The code uses two different classification algorithms, KNeighborsClassifier and RandomForestClassifier, to classify sentiments using sentence vectors. It then uses cross-validation to evaluate the performance of the two algorithms.

The cross-validation results indicate that the KNeighborsClassifier model has an average accuracy of 0.76 with a standard deviation of 0.017, while the RandomForestClassifier model has an average accuracy of 0.70 with a standard deviation of 0.02.

Finally, the code uses bag-of-words comparison to classify the sentiments using RandomForestClassifier. The results show that the average sentiment classification accuracy is 0.7373 with a standard deviation of 0.0159. This indicates that the classification model performs well on the test data and has some ability to generalize to unknown sentence data. However, the performance could be improved by using other word processing techniques or by adjusting the hyperparameters of the model.