Maksym Shevchenko, AI-1

Reuters-21578 text classification

GitHub repository: https://github.com/Gurdel/Magistracy/tree/main/Artificial%20Intelligence

Reuters-21578

Reuters-21578 Text Categorization Collection Data Set. The collection consists of 22 data files, an SGML DTD file describing the data file format, and six files describing the categories used to index the data. Some additional files, which are not part of the collection but have been contributed by other researchers as useful resources are also included. All files are available uncompressed, and in addition a single gzipped Unix tar archive of the entire distribution is available as reuters21578.tar.gz.

The Reuters-21578 collection is distributed in 22 files. Each of the first 21 files (reut2-000.sgm through reut2-020.sgm) contain 1000 documents, while the last (reut2-021.sgm) contains 578 documents. The files are in SGML format. Readers interested in more detail on SGML are encouraged to pursue one of the many books and web pages on the subject.

Dataset creation

You can find source code here:
https://github.com/Gurdel/Magistracy/tree/main/Artificial%20Intelligence/Reuter%20classification.

1) Reuters dataset parsing and new dataset creation. Firstly, I placed all input files' text to one big file input.txt. Then I split text by tags combination <TOPIC><D> to get list of records with topic. In each record I found topic (text before <\D> tag) and message (text between <TITLE></TITLE> and between <BODY></BODY> tags). Since there were many different numbers in the messages, I replaced them with the special word int. After that, I normalized the text, leaving only the words separated by a space.

```
texts = inp.split('<TOPICS><D>')
for t in texts[1:]:
    try:
        topic = t.split('</D>')[0].replace('-', '')
        title = t.split('<TITLE>')[1]
        text = text.split('</TITLE>')[0]
        text = t.split('<BODY>')[1]
        text = ' ' + title + text.split('</BODY>')[0]

        #  Видаляємо (заміняємо на пробіл):
        #text = re.sub(r'(\d+[,.]?)+', 'int', text)  #  заміняємо числа на слово int
        text = re.sub('&lt', ' ', text)  #  &lt в .sgm
        text = re.sub('&#3;', ' ', text)  #  &#3; в .sgm
        text = re.sub(r'\W', ' ', text)  #  спец символи
        text = re.sub(r'\s+\D\s+', ' ', text)  #  одиничні літери
        text = re.sub(r'\s+\D\s+', ' ', text)
        text = re.sub(r'\s+\D\s+', ' ', text)
        text = re.sub(r'\^[a-zA-Z]\s+', ' ', text)  #  не літери
        text = re.sub(r'\s+[a-zA-Z]\$', ' ', text)
        text = re.sub(r'\s+', ' ', text, flags=re.I)  #  подвійні пробіли
        text = text.lower().strip()  #  переводимо в нижній регістр
        sw.write(f'{topic}\t{text}\n')
```

2) Vectorize text. I lemmatized text using nltk.stem.WordNetLemmatizer and vectorized it using bag-of-words (CountVectorizer from sklearn.feature_extraction.text). To decrease vectors dimension I decided to choose only 2000 most frequent words, set up they minimum frequency to 10 messages and maximum frequency to 80% of all messages. Also, I deleted English stopwords.

```python
for t in X:
    lem = [wnl.lemmatize(word) for word in str(t).split()]
    lem_texts.append(' '.join(lem))

from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(max_features=2000, min_df=10, max_df=0.8,
                     stop_words=nltk.corpus.stopwords.words('english'))
#max_features - к-сть слів, які використовуються для класифікації
#min_df - мін к-сть текстів, у яких міститься слово
#max_df - макс відсоток файлів, у яких міститься слово
#stop_words - шумові слова
X = cv.fit_transform(lem_texts).toarray()
```

Select models

Text classification is a standard task for which there are many ready-made solutions. Mostly used is SVM, Naive Bayes and Logistic Regression in combination with a bag-of-words, Word2vec or Doc2vec. Since the bag of words approach is easier to understand, I decided to use it. To select the models, I decided to test the existing ones in the sklearn. The results are as follows:

SGDClassifier	0.8871268656716418
LinearSVC	0.8857276119402985
RidgeClassifier	0.8759328358208955
MLPClassifier	0.8703358208955224
SVC	0.8642723880597015
PassiveAggressiveClassifier	0.8638059701492538
LogisticRegression	0.8586753731343284
Perceptron	0.8521455223880597
RandomForestClassifier	0.8442164179104478
ensemble.ExtraTreesClassifier	0.84375
KNeighborsClassifier	0.8339552238805971
LabelPropagation	0.8288246268656716
BaggingClassifier	0.8283582089552238
ComplementNB	0.8190298507462687
GradientBoostingClassifier	0.8143656716417911
MultinomialNB	0.7779850746268657
DecisionTreeClassifier	0.7686567164179104
NearestCentroid	0.7467350746268657
BernoulliNB	0.7416044776119403
tree.ExtraTreeClassifier	0.6851679104477612
GaussianNB	0.6427238805970149
AdaBoostClassifier	0.4766791044776119
CategoricalNB	0.37406716417910446
DummyClassifier	0.18703358208955223

naive bayes
linear model
svm
neighbors
neural network
tree
dummy
ensemble
semi supervised

As we can see, SVM and Logistic Regression show best results. So, I decided to use them and Naive Bayes like third simple model.

Theory

One-vs-rest (OvR for short, also referred to as One-vs-All or OvA) is a heuristic method for using binary classification algorithms for multi-class classification.

It involves splitting the multi-class dataset into multiple binary classification problems. A binary classifier is then trained on each binary classification problem and predictions are made using the model that is the most confident.

A possible downside of this approach is that it requires one model to be created for each class. For example, three classes requires three models. This could be an issue for large datasets (e.g. millions of rows), slow models (e.g. neural networks), or very large numbers of classes (e.g. hundreds of classes).

This approach requires that each model predicts a class membership probability or a probability-like score. The argmax of these scores (class index with the largest score) is then used to predict a class.

Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. The idea is to take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the current point, because this is the direction of steepest descent. Conversely, stepping in the direction of the gradient will lead to a local maximum of that function; the procedure is then known as gradient ascent.

Instead of climbing up a hill, think of gradient descent as hiking down to the bottom of a valley. This is a better analogy because it is a minimization algorithm that minimizes a given function.

The equation below describes what gradient descent does: b is the next position of our climber, while a represents his current position. The minus sign refers to the minimization part of gradient descent. The gamma in the middle is a waiting factor and the gradient term ($\Delta f(a)$) is simply the direction of the steepest descent. $\mathbf{b} = \mathbf{a} - \gamma \nabla \mathbf{f}(\mathbf{a})$ this formula basically tells us the next position we need to go, which is the direction of the steepest descent.

A bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:

A vocabulary of known words.

A measure of the presence of known words.

SVM

Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection.

The advantages of support vector machines are:

Effective in high dimensional spaces.

Still effective in cases where number of dimensions is greater than the number of samples.

Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.

Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines include:

If the number of features is much greater than the number of samples, avoid over-fitting in choosing Kernel functions and regularization term is crucial.

SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see Scores and probabilities, below).

More formally, a support-vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks like outliers detection. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to

the nearest training-data point of any class (so-called functional margin), since in general the larger the margin, the lower the generalization error of the classifier.

SVMs are helpful in text and hypertext categorization, as their application can significantly reduce the need for labeled training instances in both the standard inductive and transductive settings.

More detailed information about SVM and equations you can find here: https://en.wikipedia.org/wiki/Support-vector_machine. Main formula of linear SVM is hyperplane equation $\mathbf{w}^T\mathbf{x} - b = 0$, where w is the (not necessarily normalized) normal vector to the hyperplane.

Realization: https://github.com/Gurdel/Magistracy/tree/main/Artificial%20Intelligence/SVM

I split dataset to train and test sets and also split train set to train and validation test five times, to implement cross-validation. After that I run 50 iteration (for 5 validations, 250 epochs in total), in each of them for each class I applied gradient descend and changed class weights according to them.

After each validation for each sample, I calculated probability of belonging to every class and choose max. After I compared model prediction with expected and got next values:

```
Validation 1 accuracy: 0.8437317784256559 Test data accuracy: 0.8526119402985075
Validation 2 accuracy: 0.9102040816326531 Test data accuracy: 0.8708022388059702
Validation 3 accuracy: 0.9469387755102041 Test data accuracy: 0.8740671641791045
Validation 4 accuracy: 0.9615160349854227 Test data accuracy: 0.8698694029850746
Validation 5 accuracy: 0.9749271137026239 Test data accuracy: 0.8722014925373134
```

Logistic Regression

Logistic regression is a classification algorithm, used when the value of the target variable is categorical in nature. Logistic regression is most commonly used when the data in question has binary output, so when it belongs to one class or another, or is either a 0 or 1.

Here, by the idea of using a regression model to solve the classification problem, we rationally raise a question of whether we can draw a hypothesis function to fit to the binary dataset. For simplification, we only concern the binary classification problem with the dataset. The answer is that you will have to use a type of function, different from linear functions, called a logistic function, or a sigmoid function.

It is important to understand that logistic regression should only be used when the target variables fall into discrete categories and that if there's a range of continuous values the target value might be, logistic regression should not be used. Examples of situations you might use logistic regression in include:

Predicting if an email is spam or not spam

Whether a tumor is malignant or benign

Whether a mushroom is poisonous or edible.

When using logistic regression, a threshold is usually specified that indicates at what value the example will be put into one class vs. the other class. In the spam classification task, a threshold of 0.5 might be set, which would cause an email with a 50% or greater probability of being spam to be classified as "spam" and any email with probability less than 50% classified as "not spam".

Although logistic regression is best suited for instances of binary classification, it can be applied to multiclass classification problems, classification tasks with three or more classes. You accomplish this by applying a “one vs. all” strategy.

For more detailed information about logistic regression, please, read here: https://en.wikipedia.org/wiki/Logistic_regression.

Realization:
https://github.com/Gurdel/Magistracy/tree/main/Artificial%20Intelligence/LogisticRegression

Like in SVM model, I also used here gradient descend and one-vs-all strategy, but I didn’t manage to optimize the algorithm well, so its execution time was very long for this dataset. Therefore, I decided to reduce the number of iterations (100 for each class versus 250 in SVM) and the number of features (500 versus 2000). The execution time has become shorter, but the classification accuracy has also decreased. Therefore, I decided not to split the dataset into training and test data, but to train on the entire available dataset. The final accuracy is 73.6735074626865%.

Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes’ theorem with the “naive” assumption of conditional independence between every pair of features given the value of the class variable. Bayes’ theorem states the following relationship, given class variable y and dependent feature vector x1 through xn:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots, x_n \mid y)}{P(x_1, \dots, x_n)}$$

Using the naive conditional independence assumption that

$$P(x_i|y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i|y),$$

for all i, this relationship is simplified to

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)}$$

Since P(x1,…,xn) is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

and we can use Maximum A Posteriori (MAP) estimation to estimate P(y) and P(xi|y); the former is then the relative frequency of class y in the training set.

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of P(xi|y).

In spite of their apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require

a small amount of training data to estimate the necessary parameters. (For theoretical reasons why naive Bayes works well, and on which types of data it does, see the references below.)

Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods. The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one-dimensional distribution. This in turn helps to alleviate problems stemming from the curse of dimensionality.

Additional explanation: https://en.wikipedia.org/wiki/Naive_Bayes_classifier

Realization: https://github.com/Gurdel/Magistracy/tree/main/Artificial%20Intelligence/NaiveBayes

According to the algorithm, I performed several classes for this program: tokenizer.py just split message by spaces and removes stop-words; trainedData.py stores words frequencies; trainer.py changes trained data frequencies for each train sample; classifier.py uses pretrained data to classify test dataset. In main.ipynb I just split dataset into train and test, train model in train data and then check model predictions accuracy. The final accuracy is 69.96268656716418%.

Conclusion

I have implemented three models for classification: SVM, Logistic Regression, and Naive Bayes. SVM and Baes showed classification accuracy comparable to ready-made solutions in sklearn (87% and 70% accordingly with light blue and violet fields in the comparative table). Logistic Regression showed the worst result in comparison with the out-of-the-box solution (74% versus 85% +, see the blue boxes in the comparison table). This is due to the decrease in the number of features and iterations to speed up the algorithm.