IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

Machine Learning

Data Encoding and Pre-processing Techniques in Machine Learning

1st Author : Tambe Sneha Deepak

2nd Author : Kad Jyoti Popat

3rd Author : Prof. Lokhande D.B.(Research Guide)

4th Author : Prof. Bombale S.P.(Research Guide)

JCEI'S Jaihind Institute Managemet and Research Kuran-Vadgao Sahani, India

Abstract: Machine learning performance is fundamentally influenced by the quality of data preprocessing this work provides an analytical overview of preprocessing strategies for textual data tokenization and bag-of-words representations are discussed as foundational techniques that transform unstructured text into high-dimensional feature vectors to retain contextual relationships lost in unigram models n-gram based vocabulary construction is examined followed by an evaluation of sparsity issues and the application of efficient sparse-matrix encodings feature-selection methods particularly chi-square statistical filtering are reviewed as mechanisms for mitigating noise arising from infrequent or weakly correlated terms additionally the study highlights the implications of information loss associated with discretization and aggregation procedures interpreted through the lens of the data-processing inequality empirical findings illustrate how excessive transformation can degrade model accuracy reinforcing the importance of minimal yet effective preprocessing pipelines.

I. Introduction

Machine learning is nothing but Data preprocessing. It is initial phase as well as site consequently indeed lives patch of cell anti individual verdict-material, all over halt minimize data basis of your samples.

Usually, data preprocessing is the activity of mapping raw data into a format that is ready to pass in the direction of machine-learning techniques. Can you assume for now too there's no uncertainty in the data too you're encoding, and you'll revisit the problem of uncertainty in later chapters. At the moment, you'll learn how you can encode measurements in a way a method can acknowledge. You too also take steps to make sure your models perforın as well as possible on your data. That can mean removing characteristic to lessen every integer till parameters your model has to learn, multiplying characteristic together before apprehend interactions between them, or even combining features in more complicated ways before apprehend correlation it’s model won't be likely to find on its own.

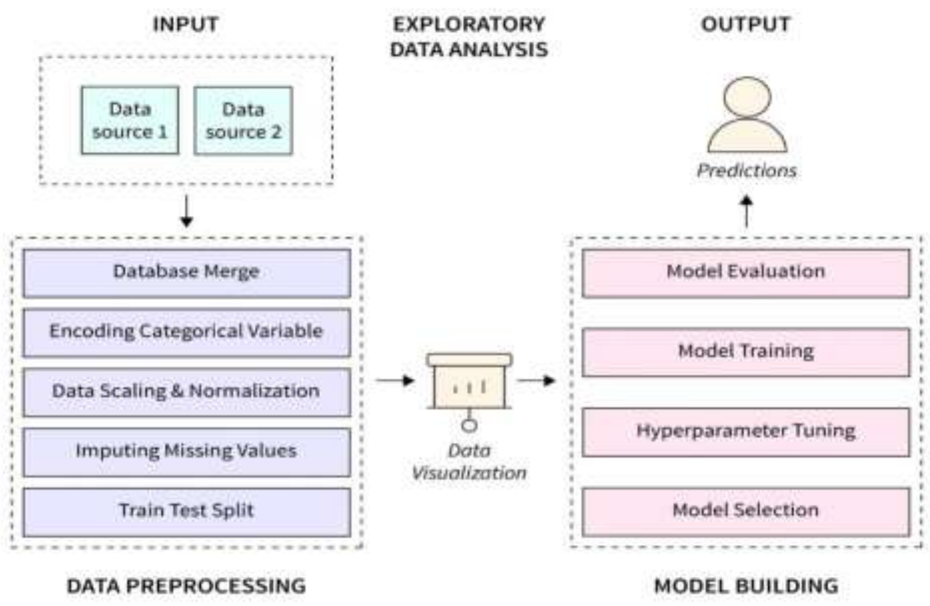
Original data

Team	Points
A	35
A	22
B	25
B	24
B	29
B	33
C	35
C	39

$\longrightarrow$

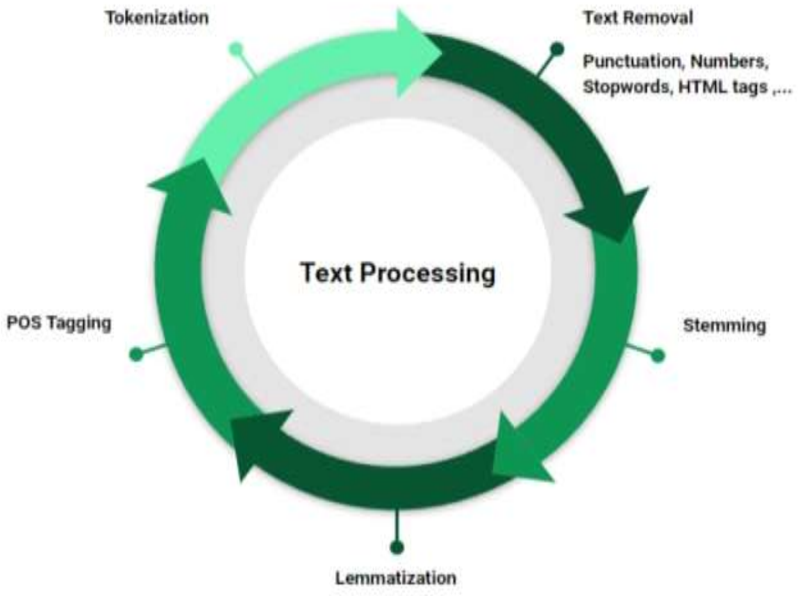
Encoded data

Team_A	Team_B	Team_C	Points
1	0	0	35
1	0	0	22
0	1	0	25
0	1	0	24
0	1	0	29
0	1	0	33
0	0	1	35
0	0	1	39



Simple text preprocessing –

The fint technique highly will accomplish the task of putting a title into a logistic regression algorithm will be tokenization. This technique lets you break up a title into terms. Then, you can encode each term like you encode the result, which lives as a binary variable if it's in the title or not. There are some strengths and weaknesses to this approach, which you'll see.



II .**Techniques used for the simple text preprocessing:**

1. Tokenization

Tokenization is a pretty simple concept. The basic idea is that you think the individual terms in a title are important for predicting the outcome, so you'll encode each one based on how much it occurs in the title.
Let's look at the title "The President vetoed the bill." You think titles with President in them might get clicked more often. The reason is that the president is important, and people want to hear about what he's doing. Youcan encode the presence of the word President as x = 1, so any title with
President in it has x = 1, and any title without it has x 0. Then, you're free to put it into the algorithm and get predictions out. You could stop here, but you notice some problems.
First, most titles aren't about the president. You'd do a lousy job if you just used a single token.
Second, not all titles about the president have the term President in then. Even if you were right that people are interested in reading about the president, maybe you should include titles that have White House, POTUS, and other related terms. Finally, we haven't really addressed how to deal with multiword terms like White House. White and House mean something very different by themselves. You'll learn how to deal with this in the next section when we talk about n-grams.
A vector written like this is often known as a bag of words since the encoding has lost the order of the terms you've jumbled them all together like taking the sentence apart and just throwing the terms in a bag you can

do a little better than this by respecting the order of terms as you'll see in the next section for now lets think more about how these words influence the mode.

There's a careful balance to strike when you're preprocessing text on the one hand the more terms you have in your vocabulary the more parameters your model needs to have in the case of logistic regression to predict a class from a piece of text you have one parameter for each term in your vocabulary if there are infrequent terms or terms with small relationships to the output you won't learn their parameters very precisely if those parameters are used in prediction they'll add more noise to the output and your model can perform worse by including them than by ignoring them in practice you might want to eliminate some terms from the vocabulary by calculating their dependence with the output like with the chi-squared test or fishers exact test you can keep the number of terms that cross-validate the best on your validation set.

2 .N-grams

A great way to respect the order of terms in a sentence is by using n-grams. The simplest version is a unigram, which is just a single-word token. The simplest nontrivial example is a bigram, which is a sequence of two words that appear consecutively in a sentence.

To see the strength in this, consider the sentence "The President lives in the White House." If you just use unigrams, you lose the ordering of White House and just consider them as the words White and House. They aren't as meaningful apart as they are together. The solution is to add a new term the vocabulary, which is the pair of those terms together.

There is still a little ambiguity. You might have the bigram and both terms that comprise it in the vocabulary! Do you encode it as having all three? Typically, you would first tokenize the text into lists of terms. Then, you'd collapse all bigrams into unigrams, usually by merging them together with an underscore. For example, White House would become ['White', 'House'] and then just [White_House]. Then, if either of White or House remains, they'll be picked up separately,

You don't normally have to implement all of this yourself. You can use packages like scikit leamy vectorizer, in sklearn, feature extraction.text.Count Vectorizer.

II.Sparsity

Once you've settled on a vocabulary that includes n-grams, you'll notice a new problem. Not every title will contain every word in the vocabulary, so the title vectors are almost all zeros! These are called sparse vectors, because there's very little (nontrivial) data in them. The problem is that they can take up a lot of computer memory. If you have 300,000 terms in your vocabulary and you have 1,000 titles, then you have 300,000,000 entries in your data matrix! That takes a lot of space for so little data!

Instead of encoding a vector as a long list of zeros with a few nonzero entries, it's much more efficient to encode them by saying which entries aren't zero. It makes handling math a little trickier, but fortunately most of this has been worked out for us! There's a great implementation of spark vector and matrix math in the scipy.sparse library.

The easiest form for constructing sparse matrices is the sparse coordinate format. Instead of a vector like [0, 1, 0, 0, 0], you would just write a list with entries like (row, column, value). In this case, we'd encode the vector as [(0. 1, 1)]. In general, you might be working with lots of vectors, so the first entry tells you which vector you're working with. Here, it's just zero. The next entry tells you which column has the nonzero entry. The last entry tells you what the value of the entry is.

The downside of this approach is that you need three values to encode a single nonzero entry in the matrix. It saves space only if less than one-third of the entries in the matrix is nonzero. This will usually be the case for reasonably large sets of documents, with reasonably large vocabularies.

There is another problem that comes up with such large numbers of columns: algorithm performance can suffer if there are too many columns of data. Sometimes it will be useful to limit to only the most important terms. You can do this by choosing a set of "features" to keep and discarding the rest it.

Consider, for example, linear regression on a large number of variables. Consider including several binary features this are completely irrelevant, so their true coefficients should be 0. If you include these variables with a finite data set, you'll generally find a value different from zero because of measurement error. Assuming a Gaussian sampling distribution for those coefficients through standard deviation, and assuming k coefficents of irrelevant variables are active for a prediction, then you'd add noise to your prediction for y with a standard deviation of sqrt(k) * sigma use the error propagation formulae for sums)! Definitely we'd be better off throwing away some of these features.

There are many ways to do this in practice. We'll talk about two: testing for dependence and using a prediction method called lasso regression.

with the first method you want to find words that correlate highly with the outcome you're trying to predict you can do this using an x 2 test since you're working with categorical two categories in this case data a good implementation is in the scikit learn also called sklearn library in sklearn feature selection select k best you can use it with the score metric sklearn feature selection chi 2 this will find the features very live most informative of the outcome and select the top k to closer wherever you specify the value of k to use.

III .Information Loss –

We've talked a lot about reducing your data in this chapter, but what we really mean is transforming your data in a way that probably loses some information.

Consider the original example, "The President vetoed the bill." You'd represent this as something the machine sees more as ['the': 2, 'President': 1, 'vetoed': 1, 'bill': 1]. If you're smart, you could guess what the sentence is saying, but it's clear you lost a lot of information when you mapped the sentence to the bag of words.

It turns out that information content is critically important when you're doing a prediction problem. If you can't reverse a data transformation, then you've lost some information. A theorem from information theory says that as you gain more information in X (that relates to Y), your prediction for Y will get better and better. The more information you lose during preprocessing, the more you set yourself back from the start. A simple example to illustrate this point is the case of tokenization.

There's another theorem from information theory called the data-processing inequality. It says that when you process data (as in preprocessing), you can end up only with less than or equal to the amount of information you started with. There are no creative ways to slice the data that will add information, unless you incorporate more data from an outside source.

Another way to lose information is by aggregating data. An aggregation is when a collection of numbers go into a calculation and a single number (or smaller collection of numbers) is the result. Common examples are surns, averages, variances, and medians.

Suppose you have a sample of website users and a statistic for each user, like the number of articles on the website that the user has read. Often, the only data that is available is that which is reported by a third party. This data is often just aggregations of the data for the individual users. Instead of the number of articles each user has read, you might have the total number of reads of articles (the sum of the individual user statistics). You might instead have the average articles read per user (the average of the user statistics).

In each of these cases, you have a single number that is calculated from a collection of numbers. Since you don't have access to the original, more granular data, you can't calculate other statistics from it. There's no way, for example, to compute the standard deviation in article views per user, so there's no way you can derive error bounds on the mean that was reported. When you can work only with aggregations of data, your ability to do more in-depth analysis can be very limited. You can no longer slice the data different ways based on user attributes. You can't even calculate other aggregations. For this reason, it's best to record data with as much granularity as you are able.

IV. Pseudo-Code and Program Logic

```
# Data Encoding and Preprocessing Pipeline for Textual Data
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_selection import chi2, SelectKBest
from sklearn.model_selection import train_test_split
from scipy import sparse
# 1. Sample Text Dataset
documents = [
    "The President vetoed the bill",
    "The President lives in the White House",
    "The bill was approved by the committee",
    "White House officials held a press briefing",
    "The committee vetoed the proposal today"
]
```

```
labels = [1, 1, 0, 1, 0]   # Example binary labels for classification
# 2. Tokenization + Bag-of-Words Encoding
vectorizer = CountVectorizer(lowercase=True)
bow_matrix = vectorizer.fit_transform(documents)
print("Vocabulary (Unigrams):")
print(vectorizer.vocabulary_)
print("\nBag-of-Words Sparse Matrix:")
print(bow_matrix)
# 3. N-grams (Unigrams + Bigrams)
ngram_vectorizer = CountVectorizer(ngram_range=(1, 2), lowercase=True)
ngram_matrix = ngram_vectorizer.fit_transform(documents)
print("\nVocabulary (Unigrams + Bigrams):")
print(ngram_vectorizer.vocabulary_)
print("\nN-gram Sparse Matrix Shape:", ngram_matrix.shape)
# 4. Sparsity Handling using Scipy Sparse Matrix
print("\nSparsity Level:")
total_elements = ngram_matrix.shape[0] * ngram_matrix.shape[1]
non_zero = ngram_matrix.count_nonzero()
print("Total Elements:", total_elements)
print("Non-Zero Elements:", non_zero)
print("Sparsity (%):", ((total_elements - non_zero) / total_elements) * 100)
# 5. Feature Selection using Chi-Square Test
k = 10   # number of best features to select
selector = SelectKBest(chi2, k=k)
X_selected = selector.fit_transform(ngram_matrix, labels)
selected_feature_indices = selector.get_support(indices=True)
selected_features = [list(ngram_vectorizer.vocabulary_.keys())[i]
                    for i in selected_feature_indices]
print("\nSelected Features (Chi-Square):")
print(selected_features)
# 6. Example of Information Loss Demonstration
# Generate continuous synthetic data
x = np.random.normal(size=1000)
y = x + 0.1 * np.random.normal(size=1000)
df = pd.DataFrame({'x': x, 'y': y})
# Original Linear Regression
from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

```
model.fit(df[['x']], df['y'])
r2_original = model.score(df[['x']], df['y'])
# Discretization + Dummy Encoding
x_bins = pd.qcut(df['x'], q=10)
x_dummy = pd.get_dummies(x_bins, drop_first=True)
model.fit(x_dummy, df['y'])
r2_discretized = model.score(x_dummy, df['y'])
print("\nR² (Original Continuous Variable):", r2_original)
print("R² (After Discretization):", r2_discretized)
```

Output –

```
Vocabulary (Unigrams):
{'the': 13, 'president': 10, 'vetoed': 15, 'bill': 1, 'lives': 8, 'in': 7, 'white': 17, 'house': 6, 'was': 16, 'approved': 0,
'by': 3, 'committee': 4, 'officials': 9, 'held': 5, 'press': 11, 'briefing': 2, 'proposal': 12, 'today': 14}
Bag-of-Words Sparse Matrix:
  (0, 13)	2
  (0, 10)	1
  (0, 15)	1
  (0, 1)	1
  (1, 13)	2
  (1, 10)	1
  (1, 8)	1
  (1, 7)	1
  (1, 17)	1
  (1, 6)	1
  (2, 13)	2
  (2, 1)	1
  (2, 16)	1
  (2, 0)	1
  (2, 3)	1
  (2, 4)	1
  (3, 17)	1
  (3, 6)	1
  (3, 9)	1
  (3, 5)	1
  (3, 11)	1
  (3, 2)	1
  (4, 13)	2
```

```
 (4, 15)	1
 (4, 4) 1
 (4, 12)	1
 (4, 14)	1
Vocabulary (Unigrams + Bigrams):
{'the': 26, 'president': 19, 'vetoed': 33, 'bill': 2, 'the president': 29, 'president vetoed': 21, 'vetoed the': 34, 'the bill': 27, 'lives': 15, 'in': 13, 'white': 37, 'house': 11, 'president lives': 20, 'lives in': 16, 'in the': 14, 'the white': 31, 'white house': 38, 'was': 35, 'approved': 0, 'by': 5, 'committee': 7, 'bill was': 3, 'was approved': 36, 'approved by': 1, 'by the': 6, 'the committee': 28, 'officials': 17, 'held': 9, 'press': 22, 'briefing': 4, 'house officials': 12, 'officials held': 18, 'held press': 10, 'press briefing': 23, 'proposal': 24, 'today': 32, 'committee vetoed': 8, 'the proposal': 30, 'proposal today': 25}
N-gram Sparse Matrix Shape: (5, 39)
Sparsity Level:
Total Elements: 195
Non-Zero Elements: 53
Sparsity (%): 72.82051282051282
Selected Features (Chi-Square):
['vetoed the', 'the bill', 'lives', 'by the', 'the committee', 'press', 'house officials', 'held press', 'today', 'committee vetoed']

R² (Original Continuous Variable): 0.9904802762911133
R² (After Discretization): 0.9377181393532461
```

V .Conclusion

The work highlights the central role of data preprocessing in determining the standard and conclusiveness of machine learning procedure specifically in a context of textual information study shows this methods such as tokenization and bag-of-words provide a practical foundation for converting unstructured text into numerical representations yet they also introduce challenges such as sparsity and the loss of word order incorporating n-grams offers a partial remedy by preserving sequences of terms though it develops the feature space and increases computational requirements.

This discussion also underscores the importance of selecting appropriate characteristics to avoid unnecessary noise and reduce dimensionality statistical approaches such as chi-square based filtering help prioritize features that present meaningfully to prediction tasks ultimately improving model performance at the same time sparse-matrix structures offer an efficient way to preserve and manage high-dimensional text information without extreme memory usage.

A key insight from this analysis is that every preprocessing step carries the potential to discard information empirical examples demonstrate that excessive transformationssuch as aggressive discretization or aggregationcan weaken model accuracy this aligns with the data-processing inequality which states that transformed data cannot contain more information than the original therefore the findings support the need for balanced preprocessing strategies enough to make the information model-ready but not so extensive that essential detail is lost.

In summary effective machine learning relies on preprocessing pipelines so minimize information loss while ensuring computational efficiency thoughtfully designed encoding vocabulary construction and feature-selection processes can significantly enhance model learning and prediction whereas overly complex transformations may hinder performance future research may focus on adaptive preprocessing techniques that respond to dataset characteristics and model requirements further refining how textual information is prepared for machine learning applications

VII .Reference

[1] Jurafsky, D. and Martin, J. H. 2023. *Speech and Language Processing*. Pearson.
[2] Han, J., Kamber, M. and Pei, J. 2019. *Data Mining: Concepts and Techniques*. Morgan Kaufmann.
[3] Forman, G. 2003. An Extensive Empirical Study of Feature Selection Metrics for Text Classification. *Journal of Machine Learning Research*, 3: 1289–1305.
[4] Kelleher, A., & Kelleher, A. (2020). Machine learning in production: Developing and optimizing data science workflows and applications. Pearson.