The Naive Bayes Classifier - Report

Author: Paweł Kozikowski

Data: January 2026

1 Introduction

Introduction of this Report is intended to outlines the underlying logic of the Naive Bayes Classifier. To start with, the Naive Bayes Classifier is build on Bayes's Theorem. The naive part comes from fundamental assumption that all features in a provided dataset are independent from each other. This situation is almost never true in reality, but works surprisingly well for classifications problems, such as SMS Spam vs Ham Classification problem.

2 System Architecture

2.1 The Data Pipeline

Technology used for that includes Polars - Python Pandas-like library but highly optimized for working on vectors. By that Polars is much more efficient comparing to Pandas. Simple structure of my Data Pipeline is presented below

```
class DataPipeline(typing.Protocol):
        def __init__(self,
                     fetcher: type[DataFetch],
                     transformer: type[DataTransform],
                     exporter: type[DataExport],
                     *args, **kwargs) -> None:
                self._fetcher: type[DataFetch] = fetcher
                self._transformer: type[DataTransform] = transformer
                self._exporter: type[DataExport] = exporter

        def fetch(self, *args, **kwargs) -> 'DataPipeline':
                ...

        def transform(self, *args, **kwargs) -> 'DataPipeline':
                ...

        def export(self, *args, **kwargs) -> typing.Optional[pl.DataFrame]:
                ...
```

2.2 KaggleHub Data Pipeline

We are working on dataset from kaggle. The idea that comes to my mind, was automatization for loading datasets from kaggle and formatting them automatically by transformer component of implemented KaggleHub Data Pipeline

```
data_pipeline = kh_pp.KaggleHubDataPipeline(
        kh_pp.KaggleHubDataFetch,
        kh_pp.SMS_SPAM_MessageDataTransformer,
        kh_pp.LinearExporter
)
```

- KaggleHubDataFetch - Component for fetching data from provided kaggle site
- SMS SPAM MessageDataTransformer - Component for 'cleaning' data that comes from Fetcher Component
- LinearExporter - Component that returns refactored data

2.3 Tokenizers

Tokenizers are used for converting text into collection of words, or group of words (depends on implementation of tokenization function). In my system there are implementations of two different tokenizers:

- Unigram Tokenizer - Split texts into single words
- Bigram Tokenizer - Split text into pair of words (Helps to find some patterns in this pairs)

Implementation of my Data Transformer enables us to use different tokenizers. Tokenizer function must be implemented according to Tokenizer function interface and correctly implement tockenize function. After implementation using different tokenizers, requires only change in Data Transformer parameter and system will automatically adapt to changes

```
class TokenizationFunction(typing.Protocol):
        @staticmethod
        def tokenize(text: typing.Optional[str], *args, **kwargs):
                ...
        @staticmethod
        def get_dtype() -> polars.DataType:
                ...
)
```

2.4 ClassifierNB

ClassifierNB is a class that represents The Naive Bayes Classifier machine learning algorithm. It enables user by simple API train model on provided data and predict classes from given input paramter.

```
class ClassifierNB:
    self.histogram = {}
    self.classes_probabilities = {}
    self.vocabluary = {}

    def fit(X, y) -> None:
        ...

    @_token
    def predict(x) -> typing.Dict:
        ...
)
```

@token, what is that? This is wrapper for our predict function for even easier API for user. Imagine situation when user has different types of input data. For example:

```
x1 = 'This is not a spam text!'
x2 = ['Hello', 'World', 'Spam']

model.predict(x1) —> predictions
model.predict(x2) —> predictions as well
```

User can provide data in tokenized or not tokenized version. @token wrapper will do tokenization automatically.

3 Mathematics

3.1 Bayes‘ Theorem

Bayes' theorem is stated mathematically as the following equation

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

Using Bayes' theorem, the conditional probability can be decomposed as:

$$p(C_k|x) = \frac{p(C_k) * p(x|C_k)}{p(x)}$$

In plain English, using Bayesian probability terminology, the above equation can be written as

$$posterior = \frac{prior * likelihood}{evidence}$$

3.2 Naive assumption

In The Naive Bayes Classifier we assume that occurrences of every token

$$x_i$$

are independent from each other. Thanks to that assumption this conditional probability can be converted into such formula:

$$P(C_k) \prod_{i=1}^{n} P(x_i|C_k)$$

3.3 Usage of logarithms

In computing multiplying many times really small real number can lead to underflow. Using logarithms saves us from this event. Why? This formula presents to us the main evidence:

$$\ln P(C_k|x) \propto \ln P(C_k) + \sum_{i=1}^{n} \ln P(x_i|C_k)$$

3.4 Laplace Smoothing - Handling Zeros

In order to avoid situation where single word that did not occure any single time during training and have probability equal to 0 we implement Laplace Smoothing:

$$P(x_i|C_k) = \frac{count(x_i, C_k) + \alpha}{count(C_k) + \alpha \cdot |V|}$$