1/19/2020

SEMEVAL 2020

Task 6

Name : Monique Ehab
ID :4928

Vectorization :

Word Counts with CountVectorizer

The CountVectorizer provides a simple way to both tokenize a collection of text documents and build a vocabulary of known words, but also to encode new documents using that vocabulary.
You can use it as follows:

1. Create an instance of the *CountVectorizer* class.
2. Call the *fit()* function in order to learn a vocabulary from one or more documents.
3. Call the *transform()* function on one or more documents as needed to encode each as a vector.

Word Frequencies with Tfidf

Word counts are a good starting point, but are very basic.

One issue with simple counts is that some words like "*the*" will appear many times and their large counts will not be very meaningful in the encoded vectors.
An alternative is to calculate word frequencies, and by far the most popular method is called **TF-IDF**. This is an acronym than stands for "*Term Frequency – Inverse Document*" Frequency which are the components of the resulting scores assigned to each word.

- **Term Frequency**: This summarizes how often a given word appears within a document.
- **Inverse Document Frequency**: This downscales words that appear a lot across documents.

Without going into the math, TF-IDF are word frequency scores that try to highlight words that are more interesting, e.g. frequent in a document but not across documents.

If you already have a learned CountVectorizer, you can use it with a TfidfTransformer to just calculate the inverse document frequencies and start encoding documents.
In order to start using TfidfTransformer you will first have to create a CountVectorizer to count the number of words (term frequency), limit your vocabulary size, apply stop words and etc. The code below does just that.

Once you have the IDF values, you can now compute the tf-idf scores for any document or set of documents through fit_transform().

Decision tree summary :

With versatile features helping actualize both categorical and continuous dependent variables, it is a type of supervised learning algorithm mostly used for classification problems. What this algorithm does is, it splits the population into two or more homogeneous sets based on the most significant attributes making the groups as distinct as possible.

However, the downside. Simple decision trees tend to over fit the training data more so that other techniques which means you generally have to do tree pruning and tune the pruning procedures. You didn't have any upfront design cost, but you'll pay that back on tuning the trees performance.

KNN :

K nearest neighbors is a simple algorithm that stores all available cases and classifies new cases based on a similarity measure (e.g., distance functions). KNN has been used in statistical estimation and pattern recognition already in the beginning of 1970’s as a non-parametric technique.

Algorithm :

A case is classified by a majority vote of its neighbors, with the case being assigned to the class most common amongst its K nearest neighbors measured by a distance function. If K = 1, then the case is simply assigned to the class of its nearest neighbor.

Distance functions

Euclidean $$\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$$

Manhattan $$\sum_{i=1}^{k}\left|x_i - y_i\right|$$

Minkowski $$\left(\sum_{i=1}^{k}(\left|x_i - y_i\right|)^q\right)^{1/q}$$

Naïve Bayes :

`MultinomialNB` implements the naive Bayes algorithm for multinomially distributed data, and is one of the two classic naive Bayes variants used in text classification (where the data are typically represented as word vector counts, although tf-idf vectors are also known to work well in practice). The distribution is parametrized by vectors $\theta_y = (\theta_{y1}, \ldots, \theta_{yn})$ for each class y, where n is the number of features (in text classification, the size of the vocabulary) and θ_{yi} is the probability $P(x_i \mid y)$ of feature i appearing in a sample belonging to class y.

The parameters θ_y is estimated by a smoothed version of maximum likelihood, i.e. relative frequency counting:

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

where $N_{yi} = \sum_{x \in T} x_i$ is the number of times feature i appears in a sample of class y in the training set T, and $N_y = \sum_{i=1}^{n} N_{yi}$ is the total count of all features for class y.

The smoothing priors $\alpha \geq 0$ accounts for features not present in the learning samples and prevents zero probabilities in further computations. Setting $\alpha = 1$ is called Laplace smoothing, while $\alpha < 1$ is called Lidstone smoothing.

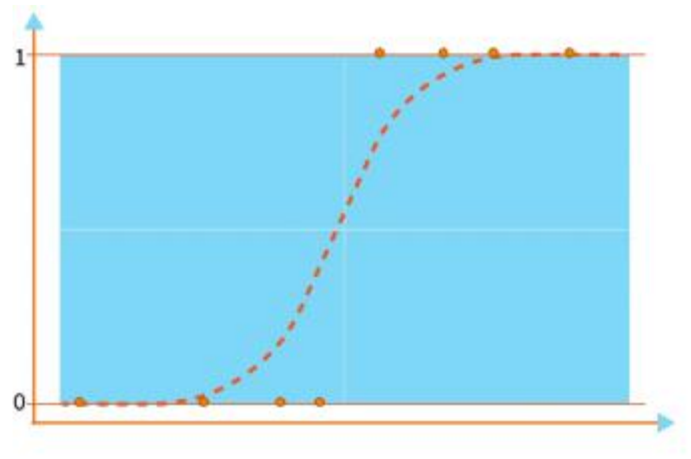
$$\underbrace{P(c \mid x)}_{\text{Posterior Probability}} = \frac{\overbrace{P(x \mid c)}^{\text{Likelihood}}\,\overbrace{P(c)}^{\text{Class Prior Probability}}}{\underbrace{P(x)}_{\text{Predictor Prior Probability}}}$$

$$P(c \mid X) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

Logistic regression :

What Is Logistic Regression?

Logistic regression is a regression technique where the dependent variable is categorical. It determines the probability using a sigmoid function



Where Bayes Excels

1. Naive Bayes is a **linear classifier** while K-NN is not; It tends to be faster when applied to big data. In comparison, k-nn is usually slower for large amounts of data, because of the calculations required for each new step in the process. If speed is important, choose Naive Bayes over K-NN.

2. In general, Naive Bayes is **highly accurate** when applied to big data. Don't discount K-NN when it comes to accuracy though; as the value of *k* in K-NN increases, the error rate decreases until it reaches that of the ideal Bayes (for k→∞).

3. Naive Bayes offers you two hyperparameters to tune for smoothing: alpha and beta. A hyperparameter is a prior parameter that are tuned on the training set to optimize it. In comparison, K-NN only has one option for tuning: the "*k*", or number of neighbors.

4. This method is not affected by the curse of dimensionality and l**arge feature sets**, while K-NN has problems with both.

5. For tasks like **robotics** and **computer vision**, Bayes outperforms decision trees.

Where K-nn Excels

1. If having **conditional independence** will highly negative affect classification, you'll want to choose K-NN over Naive Bayes. Naive Bayes can suffer from the **zero probability problem;** when a particular attribute's conditional probability equals zero, Naive Bayes will completely fail to produce a valid prediction. This could be fixed using a Laplacian estimator, but K-NN could end up being the easier choice.

2. Naive Bayes will only work if the **decision boundary** is linear, elliptic, or parabolic. Otherwise, choose K-NN.

3. Naive Bayes requires that you known the underlying **probability distributions** for categories. The algorithm compares all other classifiers against this ideal. Therefore, unless you know the probabilities and pdfs, use of the ideal Bayes is unrealistic. In comparison, K-NN doesn't require that you know anything about the underlying probability distributions.

4.K-NN (and Naive Bayes) outperform decision trees when it comes to **rare occurrences**.

Where Decision Trees Excel

1. Of the three methods, decision trees are the **easiest to explain and understand**. Most people understand hierarchical trees, and the availability of a clear diagram can help you to communicate your results. Conversely, the underlying mathematics behind Bayes Theorem can be very challenging to understand for the layperson. K-NN meets somewhere in the middle; Theoretically, you could reduce the K-NN process to an intuitive graphic, even if the underlying mechanism is probably beyond a layperson's level of understanding.

2. Decision trees have **easy to use features** to identify the most significant dimensions, handle missing values, and deal with outliers.

3. Although **over-fitting** is a major problem with decision trees, the issue could (at least, in theory) be avoided by using boosted trees or random forests. In many situations, boosting or random forests can result in trees outperforming either Bayes or K-NN. The downside to those add-ons are that they add a layer of complexity to the task and detract from the major advantage of the method, which is its simplicity.
More branches on a tree lead to more of a chance of over-fitting. Therefore, decision trees work best for a **small number of classes.** For example, the above image only results in two classes: proceed, or do not proceed.

4. Unlike Bayes and K-NN, decision trees can work directly from a **table of data,** without any prior design work.

5. If you don't know your classifiers, a decision tree will **choose those classifiers** for you from a data table. Naive Bayes requires you to know your classifiers in advance.

Where Logistic regression excels

1. Logistic Regression performs well when the dataset is linearly separable.

2. Logistic regression is less prone to over-fitting but it can overfit in high dimensional datasets. You should consider Regularization (L1 and L2) techniques to avoid over-fitting in these scenarios.

3. Logistic Regression not only gives a measure of how relevant a predictor (coefficient size) is, but also its direction of association (positive or negative).

4. Logistic regression is easier to implement, interpret and very efficient to train.

However,

5. Main limitation of Logistic Regression is the assumption of linearity between the dependent variable and the independent variables. In the real world, the data is rarely linearly separable. Most of the time data would be a jumbled mess.

6. Logistic Regression can only be used to predict discrete functions. Therefore, the dependent variable of Logistic Regression is restricted to the discrete number set. This restriction itself is problematic, as it is prohibitive to the prediction of continuous data.