**Feature Engineering**

**Learning Objectives**

* Be able to perform data transformation for categorical features, image features, and text features
* Learn best practices for deriving features, handling missing data, and automated feature engineering

**Work to Complete**

In this unit, you'll:

* Complete a series of interactive exercises
* Apply feature engineering techniques to step four of your second capstone: pre-processing and training data development

Not all data comes in a neat, tidy, numerical format. In fact, most data doesn't. To do proper machine learning, you almost always need to do some feature engineering; that is, you need to convert the data you've compiled to solve your problem into a feature matrix. This unit is divided into two subunits. In the first, you'll learn how to deal with categorical features, image features, and text features. In the second, you'll dive into derived or secondary features, handling missing data, and automated feature engineering.

### Categorical, Text, & Image Features

Data scientists regularly work with **categorical, text, and image data.** However, to execute machine learning algorithms on these data types, it's necessary to perform transformations first. Categorical data, such as the neighborhood in which a property is located, does not always work well with the machine learning algorithm you're most interested in using. Linear regression, for example, requires numerical inputs. In this subunit, you'll learn about one-hot-encoding and alternative methods for transforming data.

You'll also take a look at text and image data feature engineering, the former of which plays an important role in Natural Language Processing applications like social media data mining. Feature engineering with image data is more complex but just as important. The simplest case is using the pixel values themselves, but a plethora of more involved techniques exist and are continually evolving.

You'll learn about different tools like Sklearn and HOG (or Histogram of Oriented Gradients) that will help you perform feature engineering with these data types.

# Smarter Ways to Encode Categorical Data for Machine Learning

## Exploring Category Encoders

[[](https://jeffhale.medium.com/?source=post_page-----6dca2f71b159--------------------------------)](https://jeffhale.medium.com/?source=post_page-----6dca2f71b159--------------------------------)

[[Towards Data Science](https://towardsdatascience.com/?source=post_page-----6dca2f71b159--------------------------------)](https://towardsdatascience.com/?source=post_page-----6dca2f71b159--------------------------------)

[Jeff Hale](https://jeffhale.medium.com/?source=post_page-----6dca2f71b159--------------------------------)

·

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----6dca2f71b159--------------------------------)

·

12 min read

·

Sep 10, 2018

Better encoding of categorical data can mean better model performance. In this article I’ll introduce you to a wide range of encoding options from the [Category Encoders package](http://contrib.scikit-learn.org/categorical-encoding/index.html) for use with scikit-learn machine learning in Python.

A close-up of a typewriter

Description automatically generated with medium confidence

Enigma for encoding

# TL;DR;

Use Category Encoders to improve model performance when you have nominal or ordinal data that may provide value.

For nominal columns try OneHot, Hashing, LeaveOneOut, and Target encoding. Avoid OneHot for high cardinality columns and decision tree-based algorithms.

For ordinal columns try Ordinal (Integer), Binary, OneHot, LeaveOneOut, and Target. Helmert, Sum, BackwardDifference and Polynomial are less likely to be helpful, but if you have time or theoretic reason you might want to try them.

For regression tasks, Target and LeaveOneOut probably won’t work well.

# Roadmap

A close-up of a map

Description automatically generated

Map

In this article I’ll discuss terms, general usage and five classic encoding options: Ordinal, One Hot, Binary, BaseN, and Hashing. In the future I may evaluate Bayesian encoders and contrast encoders with roots in statistical hypothesis testing. 🚀

In [an earlier article](https://towardsdatascience.com/7-data-types-a-better-way-to-think-about-data-types-for-machine-learning-939fae99a689) I argued we should classify data as one of seven types to make better models faster. Here are the seven data types:

Useless — useless for machine learning algorithms, that is — discrete  
Nominal — groups without order — discrete  
Binary — either/or — discrete  
Ordinal — groups with order — discrete  
Count — the number of occurrences — discrete  
Time — cyclical numbers with a temporal component — continuous  
Interval — positive and/or negative numbers without a temporal component — continuous

Here we’re concerned with encoding nominal and ordinal data. A column with nominal data has values that cannot be ordered in any meaningful way. Nominal data is most often one-hot (aka dummy) encoded, but there are many options that might perform better for machine learning.

A group of trophies on a podium

Description automatically generated with low confidence

Rank

In contrast, ordinal data can be rank ordered. Ordinal data can be encoded in one of three ways, broadly speaking, but I think it’s safe to say that its encoding is often not carefully considered.

1. It can be assumed to be close enough to interval data — with relatively equal magnitudes between the values — to treat it as such. Social scientists make this assumption all the time with Likert scales. For example, “On a scale from 1 to 7, 1 being extremely unlikely, 4 being neither likely nor unlikely and 7 being extremely likely, how likely are you to recommend this movie to a friend?” Here the difference between 3 and 4 and the difference between 6 and 7 can be reasonably assumed to be similar.
2. It can be treated as nominal data, where each category has no numeric relationship to another. You can try one-hot encoding and other encodings appropriate for nominal data.
3. The magnitude of the difference between the numbers can be ignored. You can just train your model with different encodings and see which encoding works best.

In this series we’ll look at Categorical Encoders 11 encoders as of version 1.2.8. \*\*Update: Version 1.3.0 is the latest version on PyPI as of April 11, 2019.\*\*

Many of these encoding methods go by more than one name in the statistics world and sometimes one name can mean different things. We’ll follow the Category Encoders usage.

Big thanks to [Will McGinnis](http://www.willmcginnis.com/2015/11/29/beyond-one-hot-an-exploration-of-categorical-variables/) for creating and maintaining this package. It is largely derived from StatsModel’s [Patsy package](https://patsy.readthedocs.io/en/latest/API-reference.html), which in turn is based on this [UCLA statistics reference](https://stats.idre.ucla.edu/r/library/r-library-contrast-coding-systems-for-categorical-variables/).

There are an infinite number of ways to encode categorical information. The ones in Category Encoders should be sufficient for most uses. 👍

# Quick Summary

Here’s the list of Category Encoders functions with their descriptions and the type of data they would be most appropriate to encode.

## Classic Encoders

The first group of five classic encoders can be seen on a continuum of embedding information in one column (Ordinal) up to k columns (OneHot). These are very useful encodings for machine learning practitioners to understand.

**Ordinal** — convert string labels to integer values 1 through k. Ordinal.  
**OneHot** — one column for each value to compare vs. all other values. Nominal, ordinal.  
**Binary** — convert each integer to binary digits. Each binary digit gets one column. Some info loss but fewer dimensions. Ordinal.  
**BaseN** — Ordinal, Binary, or higher encoding. Nominal, ordinal. Doesn’t add much functionality. Probably avoid.  
**Hashing** — Like OneHot but fewer dimensions, some info loss due to collisions. Nominal, ordinal.  
**Sum** — Just like OneHot except one value is held constant and encoded as -1 across all columns.

## Contrast Encoders

The five contrast encoders all have multiple issues that I argue make them unlikely to be useful for machine learning. They all output one column for each value found in a column. Their [stated intents](http://www.willmcginnis.com/2015/11/29/beyond-one-hot-an-exploration-of-categorical-variables/) are below.

**Helmert** (reverse) — The mean of the dependent variable for a level is compared to the mean of the dependent variable over all previous levels.  
**Backward Difference** — the mean of the dependent variable for a level is compared with the mean of the dependent variable for the prior level.   
**Polynomial** — orthogonal polynomial contrasts. The coefficients taken on by polynomial coding for k=4 levels are the linear, quadratic, and cubic trends in the categorical variable.

## Bayesian Encoders

The Bayesian encoders use information from the dependent variable in their encodings. They output one column and can work well with high cardinality data.

**Target** — use the mean of the DV, must take steps to avoid overfitting/ response leakage. Nominal, ordinal. For classification tasks.  
**LeaveOneOut** — similar to target but avoids contamination. Nominal, ordinal. For classification tasks.  
**WeightOfEvidence** — added in v1.3. Not documented in the [docs](http://contrib.scikit-learn.org/categorical-encoding/) as of April 11, 2019. The method is explained in [this post](https://www.listendata.com/2015/03/weight-of-evidence-woe-and-information.html).  
**James-Stein** — forthcoming in v1.4. Described in the code [here](https://github.com/scikit-learn-contrib/categorical-encoding/blob/master/category_encoders/james_stein.py).  
**M-estimator** — forthcoming in v1.4. Described in the code [here](https://github.com/scikit-learn-contrib/categorical-encoding/blob/master/category_encoders/m_estimate.py). Simplified target encoder.

## Use

Category Encoders follow the same API as scikit-learn’s preprocessors. They have some added conveniences, such as the ability to easily add an encoder to a pipeline. Additionally, the encoder returns a pandas DataFrame if a DataFrame is passed to it. Here’s an example of the code with the BinaryEncoder:

We’ll tackle a few gotchas with implementation in the future. But you should be able to jump right into the first five if you are familiar with scikit-learn’s API.

Note that all Category Encoders impute missing values automatically by default. However, I recommend filling missing data data yourself prior to encoding so you can test the results of several methods. I plan to discuss imputing options in a forthcoming article, so follow [me](https://medium.com/@jeffhale) on Medium if you want to make sure you don’t miss it.

# Terminology

You might see commentators use the following terms interchangeably: dimension, feature, vector, series, independent variable, and column. I will too :) Similarly, you might see row and observation used interchangeably.

k is the original number of unique values in your data column. High cardinality means a lot of unique values (a large k). A column with hundreds of zip codes is an example of a high cardinality feature.

A red bird sitting on a branch in the snow

Description automatically generated with medium confidence

High cardinality theme bird

High dimensionality means a matrix with many dimensions. High dimensionality comes with the Curse of Dimensionality — a thorough treatment of this topic can be found [here](http://www.visiondummy.com/2014/04/curse-dimensionality-affect-classification/). The take away is that high dimensionality requires many observations and often results in overfitting.

A blue star wand on a wooden surface

Description automatically generated with medium confidence

A wand to help ward off the Curse of Dimensionality

Sparse data is a matrix with lots of zeroes relative to other values. If your encoders transform your data so that it becomes sparse, some algorithms may not work well. Sparsity can often be managed by flagging it, but many algorithms don’t work well unless the data is dense.

A picture containing nature, dune, sand dune, desert

Description automatically generated

Sparse

# Digging Into Category Encoders

Without further ado, let’s encode!

## Ordinal

OrdinalEncoder converts each string value to a whole number. The first unique value in your column becomes 1, the second becomes 2, the third becomes 3, and so on.

What the actual value was prior to encoding does not affect what it becomes when you fit\_transform with OrdinalEncoder. The first value could have been 10 and the second value could have been 3. Now they will be 1 and 2, respectively.

If the column contains nominal data, stopping after you use OrdinalEncoder is a bad idea. Your machine learning algorithm will treat the variable as continuous and assume the values are on a meaningful scale. Instead, if you have a column with values car, bus, and truck you should first encode this nominal data using OrdinalEncoder. Then encode it again using one of the methods appropriate to nominal data that we’ll explore below.

In contrast, if your column values are truly ordinal, that means that the integer assigned to each value is meaningful. Assignment should be done with intention. Say your column had the string values “First”, “Third”, and “Second” in it. Those values should be mapped to the corresponding integers by passing OrdinalEncoder a list of dicts like so:

[{"col": "finished\_race\_order",   
 "mapping": [("First", 1),   
 ("Second", 2),   
 ("Third", 3)]  
}]

Here’s the basic setup for all the code samples to follow. You can get the full notebook at [this Kaggle Kernel](https://www.kaggle.com/discdiver/category-encoders-examples).

A screen shot of a computer code

Description automatically generated with low confidence

Here’s the untransformed X column.

A screenshot of a computer

Description automatically generated with low confidence

And here’s the OrdinalEncoder code to transform the color column values from letters to integers.

A screenshot of a computer code

Description automatically generated with low confidence

All the string values are now integers.

Scikit-learn’s OrdinalEncoder does pretty much the same thing as Category Encoder’s OrdinalEncoder, but is not quite as user friendly. Scikit-learn’s encoder won’t return a pandas DataFrame. Instead it returns a NumPy array if you pass a DataFrame. It also outputs values starting with 0, compared to OrdinalEncoder’s default of outputting values starting with 1.

You could accomplish ordinal encoding by mapping string values to integers in pandas. But that’s extra work once you know how to use Category Encoders.

## OneHot

One-hot encoding is the classic approach to dealing with nominal, and maybe ordinal, data. It’s referred to as the “The Standard Approach for Categorical Data” in Kaggle’s [Machine Learning tutorial series](https://www.kaggle.com/dansbecker/using-categorical-data-with-one-hot-encoding). It also goes by the [names](https://stats.stackexchange.com/questions/308916/what-is-one-hot-encoding-called-in-scientific-literature) dummy encoding, indicator encoding, and occasionally binary encoding. Yes, this is confusing. 😉

A picture containing astronomical object, amber, sphere, universe

Description automatically generated

That’s one hot sun

The one-hot encoder creates one column for each value to compare against all other values. For each new column, a row gets a 1 if the row contained that column’s value and a 0 if it did not. Here’s how it looks:

A screenshot of a computer code

Description automatically generated with low confidence

color\_-1 is actually an extraneous column, because it’s all 0s — with no variation, it’s not helping your model learn anything. It may have been intended for missing values, but in version 1.2.8 of Category Encoders it doesn’t serve a purpose.

One-hot encoding can perform very well, but the number of new features is equal to k, the number of unique values. This feature expansion can create serious memory problems if your dataset has high cardinality features. One-hot-encoded data can also be difficult for decision-tree-based algorithms — see discussion [here](https://roamanalytics.com/2016/10/28/are-categorical-variables-getting-lost-in-your-random-forests/).

The pandas [GetDummies](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html) and scikit-learn’s [OneHotEncoder](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html) functions perform the same role as the Category Encoders OneHotEncoder. I find Category Encoders OneHotEncoder a bit nicer to use.

## Binary

Binary encoding can be thought of as a hybrid of one-hot and hashing encoders. Binary creates fewer features than one-hot, while preserving some uniqueness of values in the column. It can work well with higher dimensionality ordinal data.

A picture containing keyboard, screenshot

Description automatically generated

Binary

Here’s how it works:

* The categories are encoded by OrdinalEncoder if they aren’t already in numeric form.
* Then those integers are converted into binary code, so for example 5 becomes 101 and 10 becomes 1010
* Then the digits from that binary string are split into separate columns. So if there are 4–7 values in an ordinal column then 3 new columns are created: one for the first bit, one for the second, and one for the third.
* Each observation is encoded across the columns in its binary form.

Here’s how it looks:

A screenshot of a computer

Description automatically generated with medium confidence

The first column has no variance, so it isn’t doing anything to help the model.

With only three levels, the information embedded becomes muddled. There are many collisions and the model can’t glean much information from the features. Just one-hot encode a column if it only has a few values.

In contrast, binary really shines when the cardinality of the column is higher — with the 50 US states, for example.

Binary encoding creates fewer columns than one-hot encoding. It is more memory efficient. It also reduces the chances of dimensionality problems with higher cardinality.

For ordinal data, most values that were close to each other when in ordinal form will share many of the same values in the new columns. Many machine learning algorithms can learn that the features are similar. Binary encoding is a decent compromise for ordinal data with high cardinality.

If you’ve used binary encoding successfully, please share in the comment. For nominal data a hashing algorithm with more fine-grained control usually makes more sense.

## BaseN

When the BaseN base = 1 it is basically the same as one hot encoding. When base = 2 it is basically the same as binary encoding. McGinnis [said](http://www.willmcginnis.com/2016/12/18/basen-encoding-grid-search-category_encoders/) of this encoder, “Practically, this adds very little new functionality, rarely do people use base-3 or base-8 or any base other than ordinal or binary in real problems.”

A baseball player sliding into base

Description automatically generated with medium confidence

Base 3

The main reason for BaseN’s existence is to possibly make grid searching easier. You could use BaseN with scikit-learn’s GridSearchCV. However, if you’re going to grid search with these encoding options, you can make the encoder part of your scikit-learn pipeline and put the options in your parameter grid. I don’t see a compelling reason to use BaseN. If you do, please share in the comments.

A screenshot of a computer

Description automatically generated with medium confidence

The default base for BaseNEncoder is 2, which is the equivalent of BinaryEncoder.

## Hashing

HashingEncoder implements the [hashing trick](https://medium.com/value-stream-design/introducing-one-of-the-best-hacks-in-machine-learning-the-hashing-trick-bf6a9c8af18f). It is similar to one-hot encoding but with fewer new dimensions and some info loss due to collisions. The collisions do not significantly affect performance unless there is a great deal of overlap. An excellent discussion of the hashing trick and guidelines for selecting the number of output features can be found [here](https://booking.ai/dont-be-tricked-by-the-hashing-trick-192a6aae3087).

Here’s the ordinal column again for a refresher.

A screenshot of a computer

Description automatically generated with low confidence

And here’s the HashingEncoder with output.

A screenshot of a computer

Description automatically generated with medium confidence

The n\_components parameter controls the number of expanded columns. The default is eight columns. In our example column with three values the default results in five columns full of 0s.

If you set n\_components less than k you’ll have a small reduction in the value provided by the encoded data. You’ll also have fewer dimensions.

You can pass a hashing algorithm of your choice to HashingEncoder; the default is md5. Hashing algorithms have been very successful in some Kaggle [competitions](https://blog.myyellowroad.com/using-categorical-data-in-machine-learning-with-python-from-dummy-variables-to-deep-category-66041f734512). It’s worth trying HashingEncoder for nominal and ordinal data if you have high cardinality features. 👍

# Wrap

A pink kettlebell and running shoes

Description automatically generated with low confidence

Exercise break

That’s all for now. Here’s a recap and suggestions for when to use the encoders.

For nominal columns try OneHot, Hashing, LeaveOneOut, and Target encoding. Avoid OneHot for high cardinality columns.

For ordinal columns try Ordinal (Integer), Binary, OneHot, LeaveOneOut, and Target. Helmert, Sum, BackwardDifference and Polynomial are less likely to be helpful, but if you have time or theoretic reason you might want to try them.

The Bayesian encoders can work well for some machine learning tasks. For example, Owen Zhang used the leave one out encoding method to perform well in a [Kaggle classification challenge](https://www.slideshare.net/OwenZhang2/tips-for-data-science-competitions).

## [Data Mining](https://towardsai.net/p/category/data-mining), [Programming](https://towardsai.net/p/category/programming), [Python](https://towardsai.net/p/category/programming/python)

# Text Mining in Python: Steps and Examples

[[](https://sdhilip.medium.com/?source=post_page-----78b3f8fd913b--------------------------------)](https://sdhilip.medium.com/?source=post_page-----78b3f8fd913b--------------------------------)

[[Towards AI](https://pub.towardsai.net/?source=post_page-----78b3f8fd913b--------------------------------)](https://pub.towardsai.net/?source=post_page-----78b3f8fd913b--------------------------------)

[Dhilip Subramanian](https://sdhilip.medium.com/?source=post_page-----78b3f8fd913b--------------------------------)

·

Published in

[Towards AI](https://pub.towardsai.net/?source=post_page-----78b3f8fd913b--------------------------------)

·

7 min read

·

Aug 22, 2019

A picture containing text, monochrome, black and white, monochrome photography

Description automatically generated

In today’s scenario, one way of people’s success identified by how they are communicating and sharing information with others. That’s where the concepts of language come into the picture. However, there are many languages in the world. Each has many standards and alphabets, and the combination of these words arranged meaningfully resulted in the formation of a sentence. Each language has its own rules while developing these sentences and these sets of rules are also known as grammar.

A picture containing graphic design, child art, graphics, drawing

Description automatically generated

In today’s world, according to the industry estimates, only 20 percent of the data is being generated in the structured format as we speak, as we tweet, as we send messages on WhatsApp, Email, Facebook, Instagram or any text messages. And, the majority of this data exists in the textual form which is a highly unstructured format. In order to produce meaningful insights from the text data then we need to follow a method called Text Analysis.

# What is Text Mining?

**Text Mining is the process of deriving meaningful information from natural language text.**

A picture containing text, sketch, white, font

Description automatically generated

# **What is NLP?**

**Natural Language Processing(NLP) is a part of computer science and artificial intelligence which deals with human languages.**

In other words, NLP is a component of text mining that performs a special kind of linguistic analysis that essentially **helps a machine “read” text**. It uses a different methodology to **decipher the ambiguities in human language**, including the following: automatic summarization, part-of-speech tagging, disambiguation, chunking, as well as disambiguation and natural language understanding and recognition. We will see all the processes in a step by step manner using Python.

A picture containing text, graphics, graphic design, logo

Description automatically generated

First, we need to install the NLTK library that is the natural language toolkit for building Python programs to work with human language data and it also provides easy to use interface.

# Terminologies in NLP

## Tokenization

Tokenization is the first step in NLP. It is the process of breaking strings into tokens which in turn are small structures or units. Tokenization involves three steps which are breaking a complex sentence into words, understanding the importance of each word with respect to the sentence and finally produce structural description on an input sentence.

## **Code:**

# Importing necessary library  
import pandas as pd  
import numpy as np  
import nltk  
import os  
import nltk.corpus# sample text for performing tokenization  
text = “In Brazil they drive on the right-hand side of the road. Brazil has a large coastline on the eastern  
side of South America"# importing word\_tokenize from nltk  
from nltk.tokenize import word\_tokenize# Passing the string text into word tokenize for breaking the sentences  
token = word\_tokenize(text)  
token

## Output

['In','Brazil','they','drive', 'on','the', 'right-hand', 'side', 'of', 'the', 'road', '.', 'Brazil', 'has', 'a', 'large', 'coastline', 'on', 'the', 'eastern', 'side', 'of', 'South', 'America']

From the above output, we can see the text split into tokens. Words, comma, punctuations are called tokens.

# Finding frequency distinct in the text

## Code 1

# finding the frequency distinct in the tokens  
# Importing FreqDist library from nltk and passing token into FreqDist  
from nltk.probability import FreqDist  
fdist = FreqDist(token)  
fdist

## Output

FreqDist({'the': 3, 'Brazil': 2, 'on': 2, 'side': 2, 'of': 2, 'In': 1, 'they': 1, 'drive': 1, 'right-hand': 1, 'road': 1, ...})

‘the’ is found 3 times in the text, ‘Brazil’ is found 2 times in the text, etc.

## Code 2

# To find the frequency of top 10 words  
fdist1 = fdist.most\_common(10)  
fdist1

## Output

[('the', 3),  
 ('Brazil', 2),  
 ('on', 2),  
 ('side', 2),  
 ('of', 2),  
 ('In', 1),  
 ('they', 1),  
 ('drive', 1),  
 ('right-hand', 1),  
 ('road', 1)]

# Stemming

Stemming usually refers to normalizing words into its base form or root form.

A picture containing text, font, electric blue, logo

Description automatically generated

Here, we have words waited, waiting and waits. Here the root word is ‘wait’. There are two methods in Stemming namely, Porter Stemming (removes common morphological and inflectional endings from words) and Lancaster Stemming (a more aggressive stemming algorithm).

## Code 1

# Importing Porterstemmer from nltk library  
# Checking for the word ‘giving’   
from nltk.stem import PorterStemmer  
pst = PorterStemmer()  
pst.stem(“waiting”)

## Output

'wait'

## Code 2

# Checking for the list of words  
stm = ["waited", "waiting", "waits"]  
for word in stm :  
 print(word+ ":" +pst.stem(word))

## Output

waited:wait  
waiting:wait  
waits:wait

## Code 3

# Importing LancasterStemmer from nltk  
from nltk.stem import LancasterStemmer  
lst = LancasterStemmer()  
stm = [“giving”, “given”, “given”, “gave”]  
for word in stm :  
 print(word+ “:” +lst.stem(word))

## Output

giving:giv  
given:giv  
given:giv  
gave:gav

Lancaster is more aggressive than Porter stemmer

# Lemmatization

A picture containing text, line, font, screenshot

Description automatically generated

In simpler terms, it is the process of converting a word to its base form. The difference between stemming and lemmatization is, lemmatization considers the context and converts the word to its meaningful base form, whereas stemming just removes the last few characters, often leading to incorrect meanings and spelling errors.

For example, lemmatization would correctly identify the base form of ‘caring’ to ‘care’, whereas, stemming would cutoff the ‘ing’ part and convert it to car.

Lemmatization can be implemented in python by using Wordnet Lemmatizer, Spacy Lemmatizer, TextBlob, Stanford CoreNLP

## Code

# Importing Lemmatizer library from nltk  
from nltk.stem import WordNetLemmatizer  
lemmatizer = WordNetLemmatizer()   
   
print(“rocks :”, lemmatizer.lemmatize(“rocks”))   
print(“corpora :”, lemmatizer.lemmatize(“corpora”))

## Output

rocks : rock  
corpora : corpus

# Stop Words

“Stop words” are the most common words in a language like “the”, “a”, “at”, “for”, “above”, “on”, “is”, “all”. These words do not provide any meaning and are usually removed from texts. We can remove these stop words using nltk library

## Code

# importing stopwors from nltk library  
from nltk import word\_tokenize  
from nltk.corpus import stopwords  
a = set(stopwords.words(‘english’))text = “Cristiano Ronaldo was born on February 5, 1985, in Funchal, Madeira, Portugal.”  
text1 = word\_tokenize(text.lower())  
print(text1)stopwords = [x for x in text1 if x not in a]  
print(stopwords)

## Output

Output of text:  
['cristiano', 'ronaldo', 'was', 'born', 'on', 'february', '5', ',', '1985', ',', 'in', 'funchal', ',', 'madeira', ',', 'portugal', '.']Output of stopwords:  
['cristiano', 'ronaldo', 'born', 'february', '5', ',', '1985', ',', 'funchal', ',', 'madeira', ',', 'portugal', '.']

# Part of speech tagging (POS)

A picture containing font, text, graphics, colorfulness

Description automatically generated

Part-of-speech tagging is used to assign parts of speech to each word of a given text (such as nouns, verbs, pronouns, adverbs, conjunction, adjectives, interjection) based on its definition and its context. There are many tools available for POS taggers and some of the widely used taggers are NLTK, Spacy, TextBlob, Standford CoreNLP, etc.

## Code

text = “vote to choose a particular man or a group (party) to represent them in parliament”  
#Tokenize the text  
tex = word\_tokenize(text)  
for token in tex:  
print(nltk.pos\_tag([token]))

## Output

[('vote', 'NN')]  
[('to', 'TO')]  
[('choose', 'NN')]  
[('a', 'DT')]  
[('particular', 'JJ')]  
[('man', 'NN')]  
[('or', 'CC')]  
[('a', 'DT')]  
[('group', 'NN')]  
[('(', '(')]  
[('party', 'NN')]  
[(')', ')')]  
[('to', 'TO')]  
[('represent', 'NN')]  
[('them', 'PRP')]  
[('in', 'IN')]  
[('parliament', 'NN')]

# Named entity recognition

It is the process of detecting the named entities such as the person name, the location name, the company name, the quantities and the monetary value.

A picture containing text, font, screenshot, line

Description automatically generated

Ref: [Sujit Pal](https://www.slideshare.net/sujitpal/soda-v2-named-entity-recognition-from-streaming-test-106598233)

## Code

text = “Google’s CEO Sundar Pichai introduced the new Pixel at Minnesota Roi Centre Event”#importing chunk library from nltk  
from nltk import ne\_chunk# tokenize and POS Tagging before doing chunk  
token = word\_tokenize(text)  
tags = nltk.pos\_tag(token)  
chunk = ne\_chunk(tags)  
chunk

## **Output**

Tree('S', [Tree('GPE', [('Google', 'NNP')]), ("'s", 'POS'), Tree('ORGANIZATION', [('CEO', 'NNP'), ('Sundar', 'NNP'), ('Pichai', 'NNP')]), ('introduced', 'VBD'), ('the', 'DT'), ('new', 'JJ'), ('Pixel', 'NNP'), ('at', 'IN'), Tree('ORGANIZATION', [('Minnesota', 'NNP'), ('Roi', 'NNP'), ('Centre', 'NNP')]), ('Event', 'NNP')])

# Chunking

Chunking means picking up individual pieces of information and grouping them into bigger pieces. In the context of NLP and text mining, chunking means a grouping of words or tokens into chunks.

A picture containing text, screenshot, number, font

Description automatically generated

ref: [nltk.org](https://www.nltk.org/book/ch07.html)

## Code

text = “We saw the yellow dog”  
token = word\_tokenize(text)  
tags = nltk.pos\_tag(token)reg = “NP: {<DT>?<JJ>\*<NN>}”   
a = nltk.RegexpParser(reg)  
result = a.parse(tags)  
print(result)

## Output

(S We/PRP saw/VBD (NP the/DT yellow/JJ dog/NN))

This blog summarizes text preprocessing and covers the NLTK steps including Tokenization, Stemming, Lemmatization, POS tagging, Named entity recognition and Chunking.

Thanks for reading. Keep learning and stay tuned for more!

You can also read this article on [KDnuggets.](https://www.kdnuggets.com/2020/05/text-mining-python-steps-examples.html)

# Feature Engineering for Images: A Valuable Introduction to the HOG Feature Descriptor

[Aishwarya Singh](https://www.analyticsvidhya.com/blog/author/aishwaryasingh/) — Published On September 4, 2019 and Last Modified On April 27th, 2023

[Computer Vision](https://www.analyticsvidhya.com/blog/category/computer-vision/) [Deep Learning](https://www.analyticsvidhya.com/blog/category/deep-learning/) [Image](https://www.analyticsvidhya.com/blog/category/image/) [Intermediate](https://www.analyticsvidhya.com/blog/category/intermediate/) [Libraries](https://www.analyticsvidhya.com/blog/category/libraries/) [Programming](https://www.analyticsvidhya.com/blog/category/programming/) [Python](https://www.analyticsvidhya.com/blog/category/python-2/) [Technique](https://www.analyticsvidhya.com/blog/category/technique/) [Unstructured Data](https://www.analyticsvidhya.com/blog/category/unstructured-data/)

Feature engineering is a game-changer in the world of machine learning algorithms. It’s actually one of my favorite aspects of being a data scientist! This is where we get to experiment the most – to engineer new features from existing ones and improve our model’s performance. You would even have used various feature engineering techniques on structured data. Can we extend this technique to unstructured data, such as images? It’s an intriguing riddle for [computer vision](https://courses.analyticsvidhya.com/courses/computer-vision-using-deep-learning-version2/?utm_source=blog&utm_medium=understand-math-HOG-feature-descriptor) enthusiasts and one we will solve in this article. In this article, we will introduce you to a popular feature extraction technique for images – Histogram of Oriented Gradients, or HOG feature. We will understand what is the HOG feature descriptor, how it works (the complete math behind the algorithm), and finally, implement it in Python.

## Overview

* Learn the inner workings and math behind the HOG feature descriptor
* The HOG feature descriptor is used in computer vision popularly for object detection
* A valuable feature engineering guide for all computer vision enthusiasts

## Table of contents

* [Overview](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-overview)
* [What is a Feature Descriptor?](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-what-is-a-feature-descriptor)
* [What is Histogram of Oriented Gradients?](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-what-is-histogram-of-oriented-gradients)
* [Introduction to the HOG Feature Descriptor](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-introduction-to-the-hog-feature-descriptor)
* [Process of Calculating the Histogram of Oriented Gradients (HOG)](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-process-of-calculating-the-histogram-of-oriented-gradients-hog)
  + [Step 1: Preprocess the Data (64 x 128)](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-step-1-preprocess-the-data-64-x-128)
  + [Step 2: Calculating Gradients (direction x and y)](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-step-2-calculating-gradients-direction-x-and-y)
  + [Step 3: Calculate the Magnitude and Orientation](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-step-3-calculate-the-magnitude-and-orientation)
* [Different Methods to Create Histograms using Gradients and Orientation](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-different-methods-to-create-histograms-using-gradients-and-orientation)
  + [Method 1:](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-method-1)
  + [Method 2:](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-method-2)
  + [Method 3:](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-method-3)
  + [Method 4:](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-method-4)
  + [Step 4: Calculate Histogram of Gradients in 8×8 cells (9×1)](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-step-4-calculate-histogram-of-gradients-in-8x8-cells-9x1)
  + [Step 5: Normalize gradients in 16×16 cell (36×1)](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-step-5-normalize-gradients-in-16x16-cell-36x1)
  + [Step 6: Features for the complete image](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-step-6-features-for-the-complete-image)
* [Implementing HOG Feature Descriptor in Python](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-implementing-hog-feature-descriptor-in-python)
* [End Notes](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-end-notes)
* [Frequently Asked Questions](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/#h-frequently-asked-questions)

## What is a Feature Descriptor?

You might have had this question since you read the heading. So let’s clear that up first before we jump into the HOG part of the article.

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/article-image-1.png)

Take a look at the two images shown below. Can you differentiate between the objects in the image?

We can clearly see that the right image here has a dog and the left image has a car. Now, let me make this task slightly more complicated – identify the objects shown in the image below:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/article-image-2.png)

Still easy, right? Can you guess what was the difference between the first and the second case? The first pair of images had a lot of information, like the shape of the object, its color, the edges, background, etc.

On the other hand, the second pair had much less information (only the shape and the edges) but it was still enough to differentiate the two images.

A black background with white text

Description automatically generated with low confidence

##### Become a Full Stack Data Scientist

Transform into an expert and significantly impact the world of data science.

Do you see where I am going with this? We were easily able to differentiate the objects in the second case because it had the necessary information we would need to identify the object. And that is exactly what a feature descriptor does:

It is a simplified representation of the image that contains only the most important information about the image.

There are a number of feature descriptors out there. Here are a few of the most popular ones:

* HOG: Histogram of Oriented Gradients
* SIFT: Scale Invariant Feature Transform
* SURF: Speeded-Up Robust Feature

In this article, we are going to focus on the HOG feature descriptor and how it works. Let’s get started!

## What is Histogram of Oriented Gradients?

The Histogram of Oriented Gradients (HOG) is a popular feature descriptor technique in computer vision and image processing. It analyzes the distribution of edge orientations within an object to describe its shape and appearance. The HOG method involves computing the gradient magnitude and orientation for each pixel in an image and then dividing the image into small cells.

## Introduction to the HOG Feature Descriptor

HOG, or Histogram of Oriented Gradients, is a feature descriptor that is often used to extract features from image data. It is widely used in [computer vision](https://courses.analyticsvidhya.com/courses/computer-vision-using-deep-learning-version2/?utm_source=blog&utm_medium=understand-math-HOG-feature-descriptor) tasks for [object detection](https://www.analyticsvidhya.com/blog/2018/10/a-step-by-step-introduction-to-the-basic-object-detection-algorithms-part-1/?utm_source=blog&utm_medium=understand-math-HOG-feature-descriptor).

Let’s look at some important aspects of HOG that makes it different from other feature descriptors:

* The HOG descriptor focuses on the structure or the shape of an object. Now you might ask, how is this different from the edge features we extract for images? In the case of edge features, we only identify if the pixel is an edge or not. HOG is able to provide the edge direction as well. This is done by extracting the **gradient and orientation** (or you can say magnitude and direction) of the edges
* Additionally, these orientations are calculated in **‘localized’ portions**. This means that the complete image is broken down into smaller regions and for each region, the gradients and orientation are calculated. We will discuss this in much more detail in the upcoming sections
* Finally the HOG would generate a **Histogram** for each of these regions separately. The histograms are created using the gradients and orientations of the pixel values, hence the name ‘Histogram of Oriented Gradients’

To put a formal definition to this:

The HOG feature descriptor counts the occurrences of gradient orientation in localized portions of an image.

Implementing HOG using tools like OpenCV is extremely simple. It’s just a few lines of code since we have a predefined function called ***hog*** in the ***skimage.feature*** library. Our focus in this article, however, is on how these features are actually calculated.

## Process of Calculating the Histogram of Oriented Gradients (HOG)

We should now have a basic idea of what a HOG feature descriptor is. It’s time to delve into the core idea behind this article. Let’s discuss the step-by-step process to calculate HOG.

Consider the below image of size (180 x 280). Let us take a detailed look at how the HOG features will be created for this image:

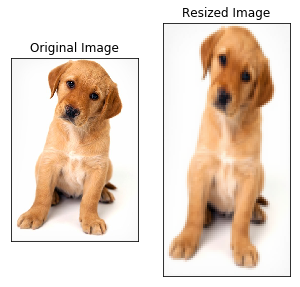
[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/puppy_image_Resize.jpeg)

### Step 1: Preprocess the Data (64 x 128)

This is a step most of you will be pretty familiar with. Preprocessing data is a crucial step in any machine learning project and that’s no different when working with images.

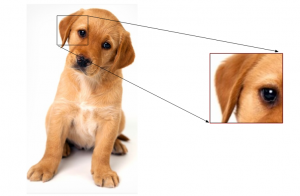
We need to preprocess the image and bring down the width to height ratio to 1:2. The image size should preferably be 64 x 128. This is because we will be dividing the image into 8\*8 and 16\*16 patches to extract the features. Having the specified size (64 x 128) will make all our calculations pretty simple. In fact, this is the exact value used in the [original paper](http://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf).

Coming back to the example we have, let us take the size 64 x 128 to be the standard image size for now. Here is the resized image:

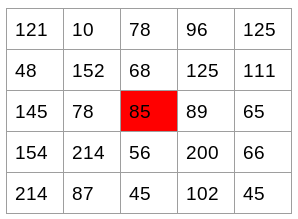
[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/index_6.png)

### Step 2: Calculating Gradients (direction x and y)

The next step is to calculate the gradient for every pixel in the image. **Gradients are the small change in the x and y directions.** Here, I am going to take a small patch from the image and calculate the gradients on that:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/Screenshot-from-2019-08-14-16-26-03.png)

We will get the pixel values for this patch. Let’s say we generate the below pixel matrix for the given patch (the matrix shown here is merely used as an example and these are not the original pixel values for the given patch):

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/article-image-3.png)

Source: [Applied Machine Learning Course](https://courses.analyticsvidhya.com/courses/applied-machine-learning-beginner-to-professional/?utm_source=blog&utm_medium=understand-math-HOG-feature-descriptor)

 I have highlighted the pixel value 85. Now, to determine the gradient (or change) in the x-direction, we need to subtract the value on the left from the pixel value on the right. Similarly, to calculate the gradient in the y-direction, we will subtract the pixel value below from the pixel value above the selected pixel.

Hence the resultant gradients in the x and y direction for this pixel are:

* Change in X direction(Gx) = 89 – 78 = 11
* Change in Y direction(Gy) = 68 – 56 = 8

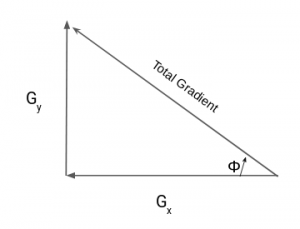
This process will give us two new matrices – one storing gradients in the x-direction and the other storing gradients in the y direction. This is similar to using a Sobel Kernel of size 1. **The magnitude would be higher when there is a sharp change in intensity, such as around the edges.**

We have calculated the gradients in both x and y direction separately. The same process is repeated for all the pixels in the image. The next step would be to find the magnitude and orientation using these values.

### Step 3: Calculate the Magnitude and Orientation

Using the gradients we calculated in the last step, we will now determine the magnitude and direction for each pixel value. For this step, we will be using the Pythagoras theorem (yes, the same one which you studied back in school!).

Take a look at the image below:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/Article-image-4.png)

The gradients are basically the base and perpendicular here. So, for the previous example, we had Gx and Gy as 11 and 8. Let’s apply the Pythagoras theorem to calculate the total gradient magnitude:

Total Gradient Magnitude =  √[(Gx)2+(Gy)2]

Total Gradient Magnitude =  √[(11)2+(8)2] = 13.6

Next, calculate the orientation (or direction) for the same pixel. We know that we can write the tan for the angles:

tan(Φ) = Gy / Gx

Hence, the value of the angle would be:

Φ = atan(Gy / Gx)

The orientation comes out to be 36 when we plug in the values. So now, for every pixel value, we have the total gradient (magnitude) and the orientation (direction). We need to generate the histogram using these gradients and orientations.

But hang on – we need to take a small break before we jump into how histograms are created in the HOG feature descriptor. Consider this a small step in the overall process. And we’ll start this by discussing some simple methods of creating Histograms using the two values that we have – gradients and orientation.

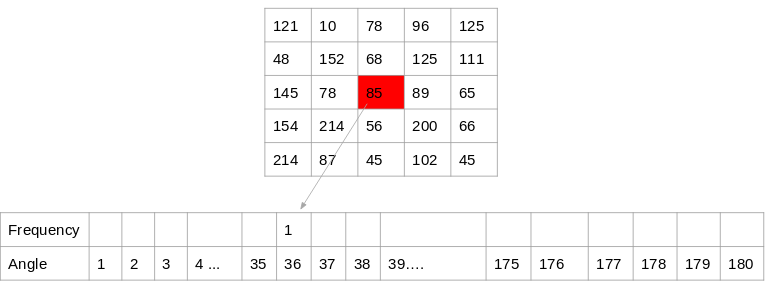
## Different Methods to Create Histograms using Gradients and Orientation

**A histogram is a plot that shows the frequency distribution of a set of continuous data.** We have the variable (in the form of bins) on the x-axis and the frequency on the y-axis. Here, we are going to take the angle or orientation on the x-axis and the frequency on the y-axis.

### ****Method 1:****

Let us start with the simplest way to generate histograms. We will take each pixel value, find the orientation of the pixel and update the frequency table.

Here is the process for the highlighted pixel (85). Since the orientation for this pixel is 36, we will add a number against angle value 36, denoting the frequency:



[Source:](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/article-image-10.png) [Applied Machine Learning Course](https://courses.analyticsvidhya.com/courses/applied-machine-learning-beginner-to-professional)

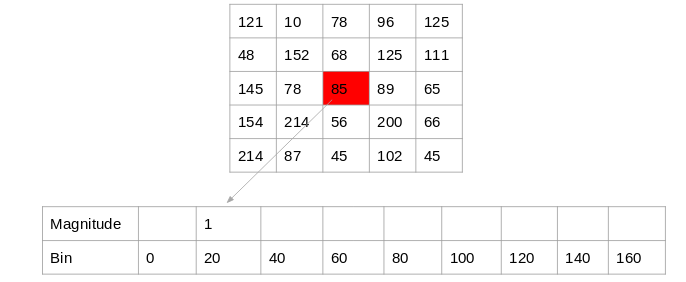
The same process is repeated for all the pixel values, and **we end up with a frequency table that denotes angles and the occurrence of these angles in the image.** This frequency table can be used to generate a histogram with angle values on the x-axis and the frequency on the y-axis.

That’s one way to create a histogram. Note that here the bin value of the histogram is 1. Hence we get about 180 different buckets, each representing an orientation value. Another method is to create the histogram features for higher bin values.

### ****Method 2:****

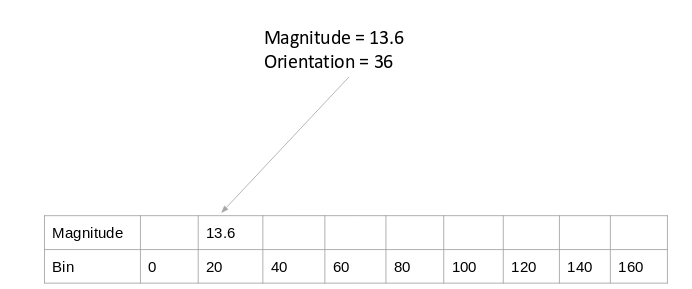
This method is similar to the previous method, except that here we have a bin size of 20. So, the number of buckets we would get here is 9.

Again, for each pixel, we will check the orientation, and store the frequency of the orientation values in the form of a 9 x 1 matrix. Plotting this would give us the histogram:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/article-image-12.png)

Source: [Applied Machine Learning Course](https://courses.analyticsvidhya.com/courses/applied-machine-learning-beginner-to-professional)

### ****Method 3:****

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/article-image-13.png)

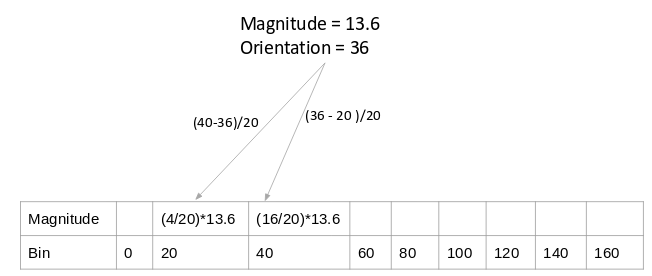
The above two methods use only the orientation values to generate histograms and do not take the gradient value into account. Here is another way in which we can generate the histogram – **instead of using the frequency, we can use the gradient magnitude to fill the values in the matrix.** Below is an example of this:

Source: [Applied Machine Learning Course](https://courses.analyticsvidhya.com/courses/applied-machine-learning-beginner-to-professional)

You might have noticed that we are using the orientation value of 30, and updating the bin 20 only. Additionally, we should give some weight to the other bin as well.

### ****Method 4:****

Let’s make a small modification to the above method. Here, we will add the contribution of a pixel’s gradient to the bins on either side of the pixel gradient. Remember, the higher contribution should be to the bin value which is closer to the orientation.

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/article-image-141.png)

Source: [Applied Machine Learning Course](https://courses.analyticsvidhya.com/courses/applied-machine-learning-beginner-to-professional)

This is exactly how histograms are created in the HOG feature descriptor.

### Step 4: Calculate Histogram of Gradients in 8×8 cells (9×1)

The histograms created in the HOG feature descriptor are not generated for the whole image. Instead, the image is divided into 8×8 cells, and the histogram of oriented gradients is computed for each cell. Why do you think this happens?

By doing so, we get the features (or histogram) for the smaller patches which in turn represent the whole image. We can certainly change this value here from 8 x 8 to 16 x 16 or 32 x 32.

If we divide the image into 8×8 cells and generate the histograms, we will get a 9 x 1 matrix for each cell. This matrix is generated using method 4 that we discussed in the previous section.

A picture containing dog breed, mammal, dog, pet

Description automatically generated

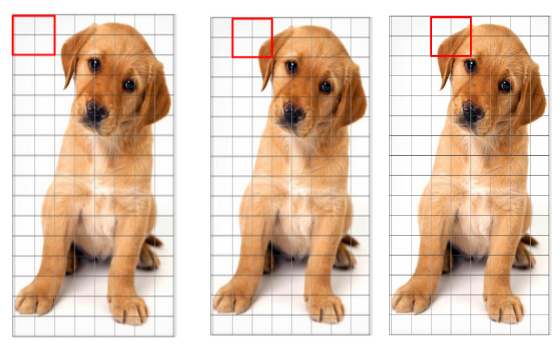
Once we have generated the HOG for the 8×8 patches in the image, the next step is to normalize the histogram.

### Step 5: Normalize gradients in 16×16 cell (36×1)

Before we understand how this is done, it’s important to understand why this is done in the first place.

Although we already have the HOG features created for the 8×8 cells of the image, the gradients of the image are sensitive to the overall lighting. This means that for a particular picture, some portion of the image would be very bright as compared to the other portions.

We cannot completely eliminate this from the image. But we can reduce this lighting variation by normalizing the gradients by taking 16×16 blocks. Here is an example that can explain how 16×16 blocks are created:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/article-image-7.png)

Here, we will be combining four 8×8 cells to create a 16×16 block. And we already know that each 8×8 cell has a 9×1 matrix for a histogram. So, we would have four 9×1 matrices or a single 36×1 matrix. To normalize this matrix, we will divide each of these values by the square root of the sum of squares of the values. Mathematically, for a given vector V:

V = [a1, a2, a3, ….a36]

We calculate the root of the sum of squares:

k = √(a1)2+ (a2)2+ (a3)2+ …. (a36)2

And divide all the values in the vector V with this value k:

A picture containing text, font, line, white

Description automatically generated

The resultant would be a normalized vector of size 36×1.

### Step 6: Features for the complete image

We are now at the final step of generating HOG features for the image. So far, we have created features for 16×16 blocks of the image. Now, we will combine all these to get the features for the final image.

Can you guess what would be the total number of features that we will have for the given image? We would first need to find out how many such 16×16 blocks would we get for a single 64×128 image:

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/article-image-8.png)

We would have 105 (7×15) blocks of 16×16. Each of these 105 blocks has a vector of 36×1 as features. Hence, the total features for the image would be 105 x 36×1 = 3780 features.

We will now generate HOG features for a single image and verify if we get the same number of features at the end.

## Implementing HOG Feature Descriptor in Python

Time to fire up Python! This, I’m sure, is the most anticipated section of this article. So let’s get rolling.

We will see how we can generate HOG features on a single image, and if the same can be applied on a larger dataset. We will first load the required libraries and the image for which we are going to create the HOG features:

|  |
| --- |
|  |
| #importing required libraries | |
|  | |

|  |
| --- |
| from skimage.io import imread, imshow |
|  |

|  |
| --- |
| from skimage.transform import resize |
|  |

|  |
| --- |
| from skimage.feature import hog |
|  |

|  |
| --- |
| from skimage import exposure |
|  |

|  |
| --- |
| import matplotlib.pyplot as plt |
|  |

|  |
| --- |
| %matplotlib inline |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| #reading the image |
|  |

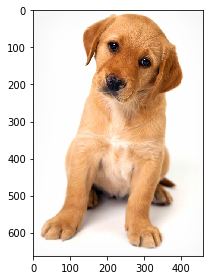
|  |
| --- |
| img = imread('puppy.jpeg') |
|  |

|  |
| --- |
| imshow(img) |
|  |

|  |
| --- |
| print(img.shape) |

[view raw](https://gist.github.com/aishwarya-singh25/d14d2dacf69f6602d8491faba42fccdc/raw/ebad8b654b5d15379dee50f3517137af8f91b4b4/reading_image.py) [reading\_image.py](https://gist.github.com/aishwarya-singh25/d14d2dacf69f6602d8491faba42fccdc#file-reading_image-py) hosted with by [GitHub](https://github.com)

(663, 459, 3)

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/index_7.png)

We can see that the shape of the image is 663 x 459. We will have to resize this image into 64 x 128. Note that we are using skimage which takes the input as height x width.

|  |
| --- |
|  |
| #resizing image | |
|  | |

|  |
| --- |
| resized\_img = resize(img, (128,64)) |
|  |

|  |
| --- |
| imshow(resized\_img) |
|  |

|  |
| --- |
| print(resized\_img.shape) |

[view raw](https://gist.github.com/aishwarya-singh25/d14d2dacf69f6602d8491faba42fccdc/raw/ebad8b654b5d15379dee50f3517137af8f91b4b4/resize_image.py) [resize\_image.py](https://gist.github.com/aishwarya-singh25/d14d2dacf69f6602d8491faba42fccdc#file-resize_image-py) hosted with by [GitHub](https://github.com)

(128, 64, 3)

## [hog_feature](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/index_8.png)

Here, I am going to use the hog function from skimage.features directly. So we don’t have to calculate the gradients, magnitude (total gradient) and orientation individually. The hog function would internally calculate it and return the feature matrix.

Also, if you set the parameter ‘visualize = True’, it will return an image of the HOG.

|  |
| --- |
|  |
| #creating hog features | |
|  | |

|  |
| --- |
| fd, hog\_image = hog(resized\_img, orientations=9, pixels\_per\_cell=(8, 8), |
|  |

|  |
| --- |
| cells\_per\_block=(2, 2), visualize=True, multichannel=True) |

[view raw](https://gist.github.com/aishwarya-singh25/d14d2dacf69f6602d8491faba42fccdc/raw/ebad8b654b5d15379dee50f3517137af8f91b4b4/hog.py) [hog.py](https://gist.github.com/aishwarya-singh25/d14d2dacf69f6602d8491faba42fccdc#file-hog-py) hosted with by [GitHub](https://github.com)

Before going ahead, let me give you a basic idea of what each of these hyperparameters represents. Alternatively, you can check the definitions from the official documentation [here](https://scikit-image.org/docs/dev/api/skimage.feature.html#skimage.feature.hog).

* The orientations are the number of buckets we want to create. Since I want to have a 9 x 1 matrix, I will set the orientations to 9
* pixels\_per\_cell defines the size of the cell for which we create the histograms. In the example we covered in this article, we used 8 x 8 cells and here I will set the same value. As mentioned previously, you can choose to change this value
* We have another hyperparameter cells\_per\_block which is the size of the block over which we normalize the histogram. Here, we mention the cells per blocks and not the number of pixels. So, instead of writing 16 x 16, we will use 2 x 2 here

The feature matrix from the function is stored in the variable fd, and the image is stored in hog\_image. Let us check the shape of the feature matrix:

|  |
| --- |
|  |
| fd.shape | |

[view raw](https://gist.github.com/aishwarya-singh25/d14d2dacf69f6602d8491faba42fccdc/raw/ebad8b654b5d15379dee50f3517137af8f91b4b4/hog_shape.py) [hog\_shape.py](https://gist.github.com/aishwarya-singh25/d14d2dacf69f6602d8491faba42fccdc#file-hog_shape-py) hosted with by [GitHub](https://github.com)

(3780,)

[A screenshot of a computer

Description automatically generated with medium confidence](https://id.analyticsvidhya.com/auth/login/?next=https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/?&utm_source=coding-window-blog&source=coding-window-blog)

As expected, we have 3,780 features for the image and this verifies the calculations we did in step 7 earlier. You can choose to change the values of the hyperparameters and that will give you a feature matrix of different sizes.

Let’s finally look at the HOG image:

|  |
| --- |
|  |
| fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 8), sharex=True, sharey=True) | |
|  | |

|  |
| --- |
|  |
|  |

|  |
| --- |
| ax1.imshow(resized\_img, cmap=plt.cm.gray) |
|  |

|  |
| --- |
| ax1.set\_title('Input image') |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Rescale histogram for better display |
|  |

|  |
| --- |
| hog\_image\_rescaled = exposure.rescale\_intensity(hog\_image, in\_range=(0, 10)) |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| ax2.imshow(hog\_image\_rescaled, cmap=plt.cm.gray) |
|  |

|  |
| --- |
| ax2.set\_title('Histogram of Oriented Gradients') |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| plt.show() |

[view raw](https://gist.github.com/aishwarya-singh25/d14d2dacf69f6602d8491faba42fccdc/raw/ebad8b654b5d15379dee50f3517137af8f91b4b4/subplot.py) [subplot.py](https://gist.github.com/aishwarya-singh25/d14d2dacf69f6602d8491faba42fccdc#file-subplot-py) hosted with by [GitHub](https://github.com)

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/08/index_9.png)

## End Notes

The idea behind this article was to give you an understanding of what is actually happening behind the HOG feature descriptor and how the features are calculated. The complete process is broken down into 7 simple steps. As a next step, we would encourage you to try using HOG features on a simple [computer vision](https://courses.analyticsvidhya.com/courses/computer-vision-using-deep-learning-version2/?utm_source=blog&utm_medium=understand-math-HOG-feature-descriptor) problem and see if the model performance improves. Do share your results in the comment section!

## Frequently Asked Questions

**Q1. What is a histogram of oriented gradients feature?**

A. A HOG (Histogram of Oriented Gradients) feature is a feature descriptor used in computer vision and image processing for object detection and recognition. Its feature descriptor represents an image’s gradient or edge orientation patterns as a histogram in machine learning models to recognize objects.

**Q2. What is HOG features in image processing?**

A. In image processing, HOG features refer to the image features extracted using the HOG algorithm. The HOG algorithm divides an image into small cells, computes each cell’s gradient orientation and magnitude, and then aggregates the gradient information into a histogram of oriented gradients. These histograms describe the image features and detect objects within an image.

**Q3. What is HOG feature for image Python?**

A. In Python, the HOG feature descriptor can be extracted using the scikit-image library, which provides functions to compute HOG features from images. The HOG feature extraction process involves specifying the histogram computation’s cell size, block size, and number of orientations.

**Q4. What is the size of HOG features?**

A. The size of HOG features depends on the parameters used in the feature extraction process, such as cell and block sizes. Generally, larger cell and block sizes will result in larger HOG features, while smaller sizes will result in smaller features. The size of the HOG features can impact the performance of machine learning models. Choosing appropriate parameters for the feature extraction process is important in object detection and recognition tasks.