Contents

1 Introduction

2 Theoretical Background (Math)

3 Nearest Centroid Classifier

The first classification algorithm we will look into is the Nearest Centroid Classidfier. It is a widely used algorithm for classification and is also one of the simplest ones to implement. We will start by looking at the mathematical background of the algorithm and then implement it in Python. This will help us to understand how to derive a classification algorithm from a mathematical model as well as the limitations of machine learning models regarding assumptions it has towards the data. Apart from this, the NNC is easy to interpret and easy to implement, which is a great starting point to get familiar with the Python programming language and the libraries we will use in this book.

3.1 Motivation

As in previous chapters mentioned, neuro scientists and other researchers used the brain as a blueprint for designing theories. If we think about the neurology that is happening when we want to do a categorization we understand on a neuronal level what happens to our brain. Of course, only to some extend for some classes of cells. And we understand how humans arrive at a certain category through psychological experiments. But we do not understand how the brain is able to do this. Between these two areas there is a huge gap. There is some research that tries to bridge this gap and ML could be a way to do this. At last by providing some Prove of Concept (PoC) to those theories. We will use a very simple psychological idea to explain the relation between the NCC and Linear Classifiers. Linear Classification is on of the most frequently used techniques in Machine Learning. Even you do it, probably on a daily basis. Now, we will bridge between the NCC and Linear Classifiers. But first we will try to understand the idea of classification through a psychological model.

Imagine you are a Neuron.
You receive a non-linear filtered sensort input $\vec{x}$, e.g. a visual input from your eyes, a smell from your nose or a noice from your ears.
How can we build abstract concepts from this information input?
In other words, *how do humans categorize different stimuli?*

For simplicity and to be able to visualize things we will imagine we only receive a 2 dimensional input.
First, we know all our data is $\vec{x} \in \mathbb{R}^2$. For example the bottom right triangle in Figure 2 could be $\vec{x} = \begin{pmatrix} 3 \\ 2.5 \end{pmatrix}$

We can also distinguish between two categories. For example, we could have a category of $\triangle$ and a category of $\bigcirc$. Now, the question is: *What do we do if we get a new data point?* For example $\vec{x}_{\times} = \begin{pmatrix} 7.5 \\ 4.0 \end{pmatrix}$

We need to find a mechanism of assigning a label to a new data point *cross*. For existing points we have this information already and we know which point

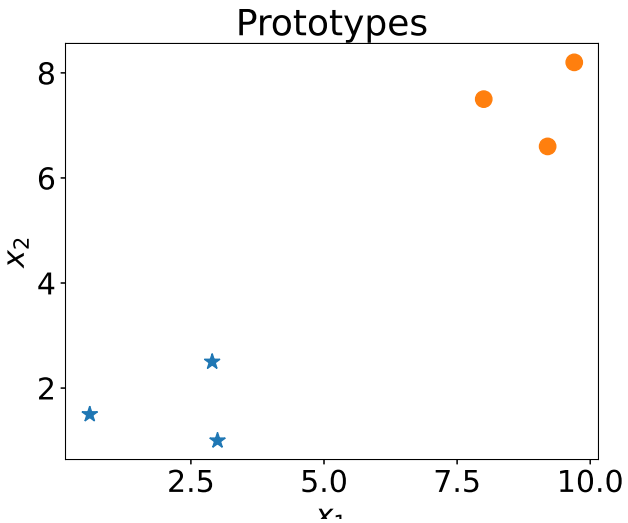

Figure 1: Prototypes of different categories

belongs to which category. So how do we know which category the new point belongs to?

Psychiologists came up with the idea of designing so called *prototypes* for each category.

This easiest solution for such prototype is calculating the mean of all points in a category. In this example we would compute the mean of all △ $\mu_\triangle$ and the mean of all ◯ $\mu_\bigcirc$.

The formula for the mean is:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{1}$$

where N is the number of samples and x_i is the i-th sample. For each of the categories this translates to

$$\mu_\triangle = \frac{1}{N_\triangle}\sum_{i=1}^{N_\triangle} x_{\triangle,i} \tag{2}$$

$$\mu_\bigcirc = \frac{1}{N_\bigcirc}\sum_{i=1}^{N_\bigcirc} x_{\bigcirc,i} \tag{3}$$

A label for a new data point $\vec{x}_\times$ can now be assigned by calculating the distance to each of the prototypes $\mu_\triangle$ and $\mu_\bigcirc$ and assigning the label of the prototype with the smallest distance.

One method to compute these distances is the **Euclidean Distance**, which is defined as

$$d(\vec{x},\vec{y}) = \sqrt{\sum_{i=1}^{N}(x_i - y_i)^2} = \sqrt{(\vec{x}-\vec{y})^T(\vec{x}-\vec{y})} \tag{4}$$

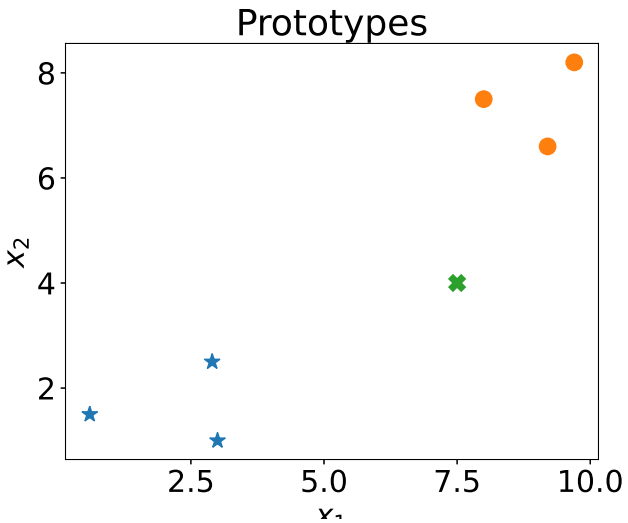

Figure 2: A new sample was added to the data set

where n is the number of dimensions of the vectors $\vec{x}$ and $\vec{y}$. There are many more distance metrics, and throughout this book we will encounter a few of them, but for now we will stick to the Euclidean Distance. After computing all distances they can be compared and the label of the prototype with the smallest distance can be assigned to the new data point.

Congratulations! You just implemented your first classification algorithm, the Nearest Centroid Classifier.

3.2 Implementation

In the following, we will look into different implementations of the NCC algorithm. And will look into two different approaches to compute the prototypes. These two approaches can be used in most common Machine Learning algorithms.

3.2.1 Inference

The pseudo code to perform a classification (inference) using the NCC algirithm is very simple:

Data: $\vec{x} \in \mathbb{R}^D, \mu_k, k \in \{1, \ldots, K\}$
Result: k^*
`# Compute nearest class centroid`
1 $k^* \leftarrow \text{argmin}_{k \in \{1,\ldots,K\}} d(\vec{x}, \mu_k)$;

Algorithm 1: Nearest **C**entroid **C**lassifier Inference

There are different ways to compute the centroids, here we used the means. To compute the means we can use two different approaches, we call these approaches *batch* and *streaming*. These terms might not 100% match the common

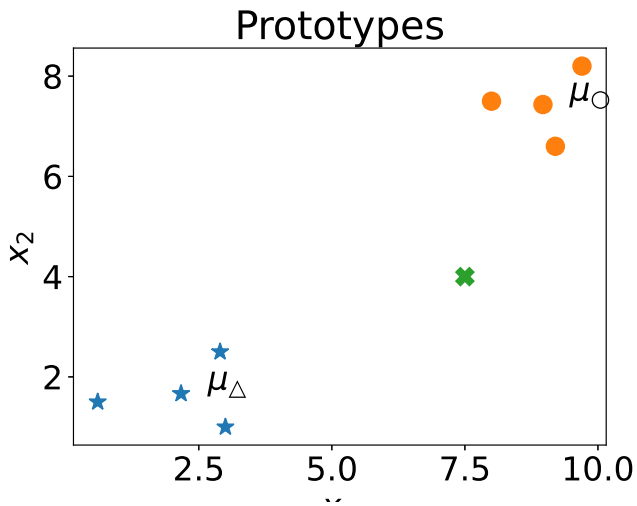

Figure 3: Mean values as prototypes for different categories

understanding of these terms, but we will use them to distinguish between the two approaches.

Batched refers to the fact that we compute the mean of all samples in a category at once. This approach requires us to store all samples in memory and then compute the mean. This is the easier, but more expensive approach.

Streaming refers to the fact that we compute the mean of all samples in a category one by one. This approach does not require us to store all samples in memory and is therefore more memory efficient. This approach is also called *online* learning.

3.2.2 Batched

The batched approach is the easier one to implement. We simply store all samples in memory and then compute the mean.

Data: $\vec{x} \in \mathbb{R}^D, \mu_k, k \in \{1, \ldots, K\}$
Result: means $\mu_k, k \in \{1 \ldots k\}$
`# Init means and counters for each class`
`# Computation of class means`
1 **for** *class* k *in* K **do**
2 $\quad \mu_k \leftarrow \frac{1}{N_k} \sum_{i=1}^{N_k} \vec{x_i}$;
3 **end**

Algorithm 2: NCC Means (Batched)

3.2.3 Streaming

To derive the streaming approach we need to look at the mathematical definition of the batched version

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{N_k} \vec{x_i} \tag{5}$$

Imagine now, that we don't actually have the N-th data point yet. We can rewrite the equation as

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{N_k-1} \vec{x_i} + \frac{1}{N_k} \vec{x_N} \tag{6}$$

We can see the factor $\frac{1}{N_k}$ is the same for all terms. This factor can be rewritten a bit differently

$$\frac{1}{N_k} = \frac{1}{N_k - 1} \cdot \frac{N_k - 1}{N_k} \tag{7}$$

Now we can rewrite the equation as

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{N_k-1} \vec{x_i} + \frac{1}{N_k} \cdot \vec{x_N} = \frac{N_k - 1}{N_k} \frac{1}{N_k - 1} \sum_{i=1}^{N_k-1} \vec{x_i} + \frac{1}{N_k} \cdot \vec{x_N} \tag{8}$$

If we compare the middle term $\frac{1}{N_k} \sum_{i=1}^{N_k-1} \vec{x_i}$ with the original equation (1) we can see that it is just μ_{k-1}.

$$\Rightarrow \frac{N_k - 1}{N_k} \mu_{k-1} + \frac{1}{N_k} \cdot \vec{x_k} \tag{9}$$

This equation can be used to iteratively compute the mean of a category. We can start with $\mu_0 = \vec{0}$ and then compute the mean of each sample by using the equation (9).

Data: $\vec{x} \in \mathbb{R}^D$ labels $y_1, \ldots, y_N \in \{1, \ldots, K\}$
Result: means $\mu_k, k \in \{1 \ldots k\}$
`# Init means and counters for each class`
1 $\forall k : \mu_k \leftarrow \vec{0}, N_k = 0$;
2 **for** *Data point* $i = 1, \ldots, N$ **do**
`# Update means and counters`
3 $k \leftarrow y_i$;
4 $\mu_k \leftarrow \frac{N_k}{N_k+1} \mu_k + \frac{1}{N_k+1} \cdot \vec{x_i}$;
5 $N_k \leftarrow N_k + 1$;
6 **end**

Algorithm 3: NCC Means (Streaming)

3.3 Limitations

Once we implemented on of the two approaches we can use it to classify new data points. For the example above this might result in *Decision Boundaries* as shown in Figure 4.

The NCC is a very simple algorithm and therefore has some limitations.

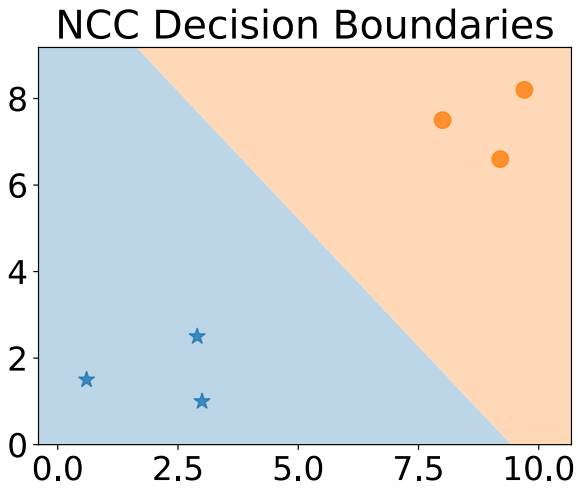

Figure 4: Decision Boundaries

- The NCC is a linear classifier. This means it can only separate linearly separable data.
- The NCC is very sensitive to outliers. If there are outliers in the data set, the mean will be shifted towards the outlier.
- The NCC is very sensitive to the number of samples in each category. If there are more samples in one category than in another, the mean will be shifted towards the category with more samples.

Let us look at the first limitation, the NCC is a linear classifier. This means it can only separate linearly separable data, similar to the data in Figure 4. If we have data that is not linearly separable, the NCC will not be able to separate it. An example of such data is shown in Figure 5.

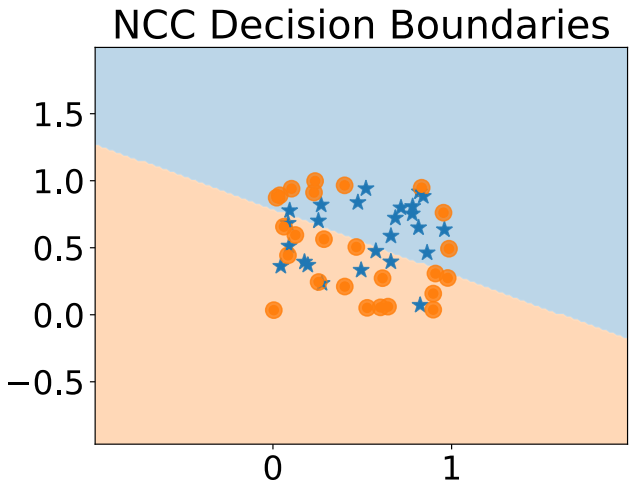

Figure 5: Non-linearly separable data

4 Nearest Neighbor Classifier

5 Hyperparameter Optimization

6 Feature Extraction

In this chapter we will talk about feature extraction, which is an essential part of any ML project/workflow. We will introduce these by looking at a simple example and then look at some more advanced techniques. After we covered most common approaches we will look into a very important concept, Machine Learning Pipelines.

6.1 Motivation

Feature Extraction describes the transformations from any kind of data to vectors. Until now, we always assumed to have a vector representation of our training data. We used the notation of $\vec{x} \in \mathbb{R}^d$ to describe a d-dimensional data point. To represent the full data set we used the convention $\vec{X} \in \mathbb{R}^{N \times d}$ a N-by-d matrix, where each row represents a data point. This notation is commonly used among ML practitioners and widely implemented in ML libraries. But it is not the only way to represent data. In fact, it is not even the most common way to represent data. In this chapter we will look into different ways to represent data and how to transform data into a vector representation.

One of the most important things in solving a problem with an ML model is selecting correct features. Most moder achievements in ML are not due to new algorithms, but due to better ways to extract features. Thankfully, for most problems we do not need to use any fancy feature extraction techniques or bleeding edge research. Classical, standard feature extraction techniques are sufficient to solve most problems in a satisfying manner. In this chapter we will look into some of these techniques, but there are many more. It is important to underline here, that often the best feature extractors are created by experts with domain knowledge. Sometimes this expert can be you, but often it is not. In this case it is important to talk to the experts and understand the problem and challenges. But for many problems and data types there are prebuild methods that we can apply without aquiring the domaing knowledge. Imagine any work in the Natural Language Processing (NLP) field. It would be horribly bad if every ML practitioner would need to study linguistics prior to working on an NLP problem. Lastly, many of theses feature extraction techniques have to be optimized in order to achieve the best results possible. The easiest approach for this is trial and error.

In this chapter we look into four different types of data and different techniques how to extract features from them.

6.2 Continuous Features

Continuous features are the easiest to understand and to work with. They are also the most common type of data. Continues data, is simply numerical data as real or integer values $x \in \mathbb{R}^d$. In many models continuous features are not required to be transformed, because they can be used directly. But for some models it is benefitial to normalize continuous features. For instance if we optimize our model with gradient descent (GD) or when we apply regularization to our model[1].

[1] Both of these ideas will be introduced in the future, but it is important to mention them here.

Given a feature $\vec{x} \in \mathbb{R}^d$ (analog for multivariant) there are several standard normalization options.

6.2.1 Normalization: z-Score

One of the normalization methods foro continuous features is the z-Score or standard scaling. The z-Score is defined as

$$z = \frac{x - \mu}{\sigma} \tag{10}$$

where μ is the mean and σ is the standard deviation of the feature.

In Python we can implement this as follows:

Listing 1: z-Score in Python

```
def z_score(X):
  return (X - X.mean(axis=0)) / X.std(axis=0)
```

Imagine the data set $\vec{X} \in \mathbb{R}^{N \times 1}$ (N is the number of samples), then we can compute the mean of $\vec{X}$ through X.mean(axis=0). Analogously this approach works for the multivariant case $\vec{X} \in \mathbb{R}^{N \times d}$. The same applies for the standard deviation X.std(axis=0). Putting everything together we get the z-Score for a data set $\vec{X}$ as implemented in Code 1. This method is also implemented in sklearn.preprocessing.StandardScaler.

6.2.2 Normalization: Min-Max-Scaling

Another form of normalization is Min-Max-Scaling. The previous normalization method is great if your data is in the shape of a normal distribution. If it isn't, you might as well chose Min-Max-Scaling, which ensures that the minimum values $\min(\vec{x})$ and maximum values $\max(\vec{x})$ of the scaled data are in a certain range, e.g. $[0, 1]$. It does so by computing the min $\min(\vec{x})$ and max $\max(\vec{x})$ of the feature and then scaling the data as follows

$$x_{scaled} = \frac{x - \min(\vec{x})}{\max(\vec{x}) - \min(\vec{x})} \tag{11}$$

The resulting variable is in the range $[0, 1]$ or any other. This method is also implemented in sklearn.preprocessing.MinMaxScaler. I leave the implementation as an exercise to the reader.

6.3 Categorical Features

As mentioned earlier continues features often don't need to be trasnformed. But categorical features most certainly need to be transformed. Categorical features are variables $x \in C$ where C can be any finite set of N values without implicit ordering, e.g.

- $C = \{red, green, blue\}$
- $C = \{dog, cat, mouse, horse\}$
- $C = \{1, 2, 6, 4, 5, 3\}$

- $C \in \{\text{User.id}\}$

The last example is a bit special, because it is not a finite set. But it is still a categorical feature, because it is not a continues feature. The first three examples are called *nominal* categorical features, because there is no implicit ordering. To use these features in a ML model we need to transform them into a vector representation. Here we will introduce the technique of *One-Hot-Encoding* (OHE) to transform categorical features into a vector representation, but there are also other techniques like for neural networks so called *Embeddings*.

6.3.1 One-Hot-Encoding

We assume we have a fixed set of categorical values, e.g. $C = \{red, green, blue\}$. Then to generate the one-hot-encoding we first need to compute the cardinality of C. The cardinality of a set is the number of elements in the set. In our example the cardinality of C is $|C| = 3$. This means we need to transform each categorical value into a row-vector of length $|C|$. For each categorical value we create a vector of length $|C|$ and set the value of the corresponding index to 1 and all other values to 0, so that

- $red \rightarrow \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$
- $green \rightarrow \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}$
- $blue \rightarrow \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$

By doing that categories can be easily represented as vectors. This is also implemented in sklearn.preprocessing.OneHotEncoder. An example for using this approach are bag-of-words vectors, which we will look into in a second.

6.3.2 One-hot-Encoding: Problems

One of the main problems of OHE is that the cardinality of the categorical feature needs to be estimated upfront, i.e. prior to creation of the OHE vectors. Therefore new items/categories can not be represented. Moreover, we lose the information on similarity between the categories, e.g. "light-blue" is as different as "green" to "blue".

Note In many situations we will face mixtures of continuous and categorical features. Usually we need to apply different extraction methods to single features to put together a general feature representation of a sample.

6.4 Text Features

The next data type we will look into is text data. Again, we have several options here and especially lately this data type enjoys great popularity and research. We will look at one approach more closely and shortly touch a few others. The approach we will focus on is Bag-of-Words (BoW), a simple yet still very powerful technique that is very similar to the one-hot-encoding.

6.4.1 Bag-of-Words Features

To create BoW-Features we count the word occurrences in the given text. They are basically histograms of word occurrences.

Example Imagine we have the following text

"The tokenizer splits the text into tokens."

Then we can create a BoW-Feature vector as follows

Word	the	tokenizer	splits	text	into	tokens	.
Count	2	1	1	1	1	1	1

Table 1: BoW-Feature vector

How does this work?

1. split the text into "tokens", e.g. words
2. build one-hot-like vector for all found words
3. count the occurrences of each word inside the text

In the BoW approach x is the vector of word occurrences. The name of an dimension is the word assigned to it. $\vec{x} \in \mathbb{N}^d$ where d is the number of unique words, the word vocabulary, in the text. d needs to be determined prior to starting and can become quite big, which we will discuss in the following.

6.4.2 Bag-of-Words: Problems

BoW-Features only account for the word historgram and due to that most of the natural language structure is lost. We lose for example the order of words, or context when processing multiple senteces at once.

Overall, BoW-Features are still very efficient and a reasonable approach.

6.4.3 Bag-of-Words: Model with N-Grams

To overcome the issue of losing language structure we can apply a different way of tokenizing our text document. Before we used word-tokens, with N-Grams we use sequences of words, specifically sequences of N words. E.g. Bi-Grams constst of two words If we would consider every combination of

Word	the tokenizer	tokenizer splits	splits the	the text	text into	into tokens	tokens .
Count	1	1	1	1	1	1	1

Table 2: BoW-Feature Bi-Gram vector

words the dimensionality of this new vector $\vec{x}$ will also significantly change, because we represent sequences of words instead of single words now. For the Bi-Gram example we get $x \in \mathbb{N}^{d^2}$ where d is the number of unique Bi-Grams in the text. An implementation of BoW-Features is available in sklearn's sklearn.feature_extraction.text.CountVectorizer.

6.4.4 Bag-of-Words: Character N-Grams

We will briefly look into another approach to BoW-Features, which is using character N-Grams instead of word N-Grams. This approach uses character sequences instead of word sequences. This approach is especially useful for languages with a rich morphology, e.g. German. Another great application is for language independent models, e.g. for language detection. Eseentially, we can select a tokenization method that fits our needs best, then transform these tokens into a BoW encoding.

6.4.5 Term Frequency - Inverse Document Frequency (TF-IDF)

One of the problems of using token occurrences is that natural language contains a lot of words that don't transfer information. Frequently occurring words like "the" are often not meaningful and will be weighted down by the inverse document frequency. Usually, we see a word-frequency distribution in the shape of the distribution shown in Figure 6.

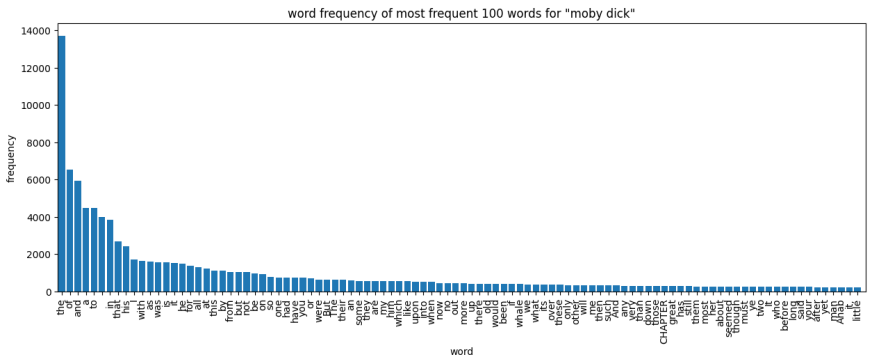

Figure 6: Word frequency distribution from Moby Dick (https://www.gutenberg.org/ebooks/2701)

In order to downweight the frequencly occurring workds to be able to only select relevant words, we aplly the inverse frequency

$$\text{idf}(t, d) = \log \frac{|\text{Documents}|}{|\text{Documents containing } t|} \tag{12}$$

A model incorporating this idf metric is TFIDF which is essentially a BoW model with applied term frequency and idf.

In sklearn you can find it under sklearn . feature_extraction . text . TfidfVectorizer .

6.4.6 Stop-Words

Another alternative to remove noice from our data are Stop-Words. Instead of computing tf and idf, we have a set of predefined words to exclude from the document. For most languages we can do that and prepare a set of words that won't change frequently. From my experience they seem to work great, and I use them more often than TF-IDF.

6.4.7 N-Grams are expensive

We quickly jump back to the N-Gram tokens and discuss the dimensionality of our BoW vectors.

If we use unigrams ($N = 1$) the size of our vectors $\vec{x} \in \mathbb{R}^d$ is d the size of our vocabulary. The size of this is v in the worst case. Libraries use so called sparse vectors to represent these high dimensional data points, dense representations require too much memory by storing 0 values. When we increase N to $N = 2$ (Bi-Grams) this memory requirement grows by a factor of d. So the worst case scenario increases to $d^2 \Rightarrow \vec{x} \in \mathbb{R}^{d^2}$. But mostly we do not gain much from this, in terms of increasing the context that is encoded. This applies annalogously to higher orders of N-Grams, e.g. $N = 3$ (Tri-Grams) require V^3 dimensions and so on. If we consider e.g. German as a language, and we know that the vocabulary for this language is somewhere in the realm of $d \approx 10^{12}$. This will cause big memory issues, especially when using anything else but Uni-Grams.

Before neural networks and deep learning google provided big sets of N-Gram data for many languages, so no one must create them themself. You can find up to $N = 5$-Grams here: `http://storage.googleapis.com/books/ngrams/books/datasetsv2.html`.

6.4.8 Hashed N-Grams

Another optimization trick to reduce memory of N-Grams is using hash-maps.

For those who know about hash-maps, instead of representing words/tokens as vectors, each dimension corresponds to a hash bucket using a hash function. This can cause hash-collision, a problem where two or more values are hashed to the same hash-value. It appears when the number of hash-buckets is lower than the number of N-Grams we want to store. But because of the structure of natural language we rarely have multiple senteces containing the same words in different orders, the collision rate is low to insignificant. I highly encourage the reader to try it out, if you run out of memory with regular vectorizers.

6.5 Image Features

We discussed until now continuous and categorical features, as well as text features. To complete this we will briefly look at methods to extract features from images. Due to the complexity of these methods we will only motivate them, because thoroughly understanding them would go beyond the scope of this book. Most of them are very easy to use, but hard to understand.

6.5.1 Classical Computer Vision

Before 2012 (ImageNet Moment) researches used classic CV methods to extract features from images. Mostly Furrier Decomposition was applied, this measures spacial frequencies of patches of the image. The resulting frequency strength of each patch is then used as a feature. These spacial frequencies are gradients in the image. High spacial frequencies are edges, and the phase of the frequency is its orientation. These kind of features are computed in the Histogram of Oriented Gradients/Edges (HOG) feature extractor, as showcased in Figure 7.

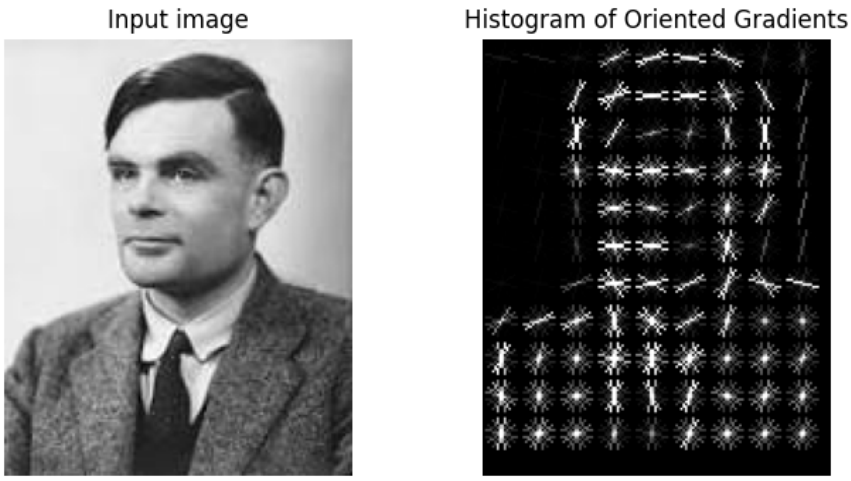

Figure 7: HOG feature extraction

HOG suits very well for object detection, because it is very robust to changes in illumination and other factors. For this reason it is widely used in e.g. self-driving cars for pedestrian recognition. They work good if we only want to encode a shape, e.g. a pedestrian, but not for more complex tasks like face recognition. In skimage you can find the HOG extractor in skimage.feature.hog.

6.5.2 Convolutional Neural Networks

Another more modern way of dealing with images is using Convolutional Neural Networks (CNNs). They are state-of-the-art (SOTA) for image classification and object detection. We don't have much time to look detaied into them, but the key takeaway here is:

We won't train these models from scratch. We will apply pretrained models, because we won't be able to achieve same performance with models we would train ourselves. And we don't want to spend so much time and money on training these models. Thankfully, we don't need to because there are many pretrained models publicly available. These models are trained foro a long time on very large datasets, e.g. ImageNet[1], for us! All we have to do is to download these models and run them as feature extractors. You can read more about CNNs in my these [2].

7 Machine Learning Pipelines

Recall the pipeline diagram from the first chapter.

Up until now we always assumed to have a vector representation $\vec{x} \in \mathbb{R}^d$ of our data. Starting from now, our data might be in any other format, like text or images.

To build a system that gets a certain format of input, transform this input into d dimensional vectors and feeds them into our model. We will now look at how to program such an ML-Pipeline

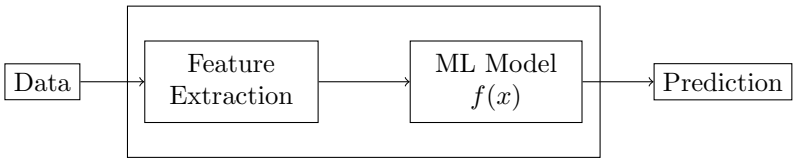

Figure 8: Machine Learning Pipeline

7.1 Motivation

In the previous chapters we looked into different feature extraction techniques. We also looked in previous chapters into different ML models and how to train them. Now, we will look into how to combine these two concepts into a single pipeline.

One way how to implement this is to manually write code that executes all steps of the pipeline sequentially. But this would be quite static and not very flexible. We would need to rewrite the code for every new data set or whenever we want to change a part of the pipeline.

A better way to implement this is to use a so called *Pipeline* object. This object is a wrapper around all steps of the pipeline. The most popular API-Interface for this is implemented by the guys from scikit-learn [**?**].

We will look into the implementation of such a pipeline in the following because it will allow to combine own implementations with implementations inside scikit-learn. This is especially useful because scikit-learn delivers plenty of tools, feature extraction algorithms and model architectures.

I personally work a lot with scikit-learn and I highly recommend it to anyone who wants to get started with ML as well assigned encourage everyone to contribute to the project.

7.2 Estimators

8 Metrics

9 Model Evaluation

10 Perceptron

11 Decision Trees

12 Regression

13 Principal Component Analysis (PCA)

14 Clustering

15 Neural Networks

References

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[2] P. Zettl, "Machine learning methods for localization and classification of insects in images," 2022.