Contents

Chapter 1

Introduction

TODO:

Chapter 2

Theoretical Background (Math)

TODO:

Chapter 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 3.2 could be $\vec{x} = \begin{pmatrix} 3 \\ 2.5 \end{pmatrix}$

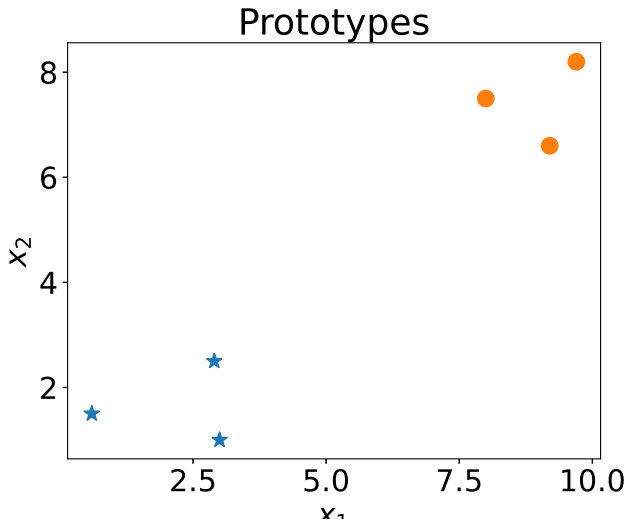


Figure 3.1: Prototypes of different categories

We can also distinguish between two categories. For example, we could have a category of △ and a category of ◯. 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 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 △ $\vec{\mu}_\triangle$ and the mean of all ◯ $\vec{\mu}_\bigcirc$.

The formula for the mean is:

$$\vec{\mu} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{3.1}$$

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

$$\vec{\mu}_\triangle = \frac{1}{N_\triangle} \sum_{i=1}^{N_\triangle} x_{\triangle,i} \tag{3.2}$$

$$\vec{\mu}_\bigcirc = \frac{1}{N_\bigcirc} \sum_{i=1}^{N_\bigcirc} x_{\bigcirc,i} \tag{3.3}$$

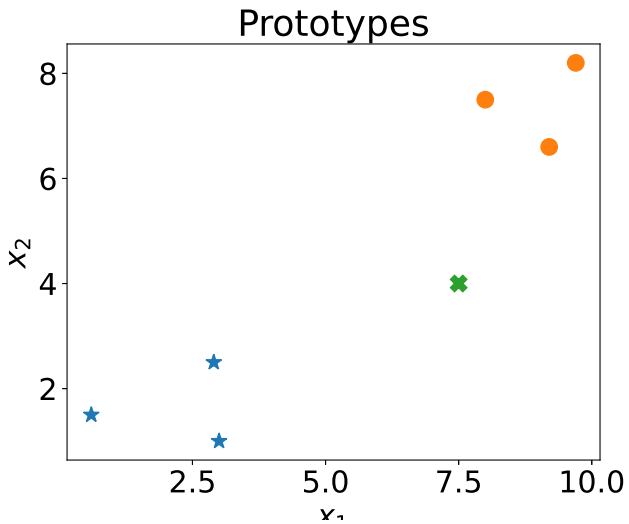


Figure 3.2: A new sample was added to the data set

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

$$d(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{N}(x_i - y_i)^2} = \sqrt{(\vec{x} - \vec{y})^T(\vec{x} - \vec{y})} \tag{3.4}$$

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. Mathematically this can be written as

$$k^* = \underset{k \in \{1,\dots,K\}}{\operatorname{argmin}} \; d(\vec{x}, \vec{\mu}_k) \tag{3.5}$$

with k^* being the label of the new data point $\vec{x}$ and K being the number of categories.

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.

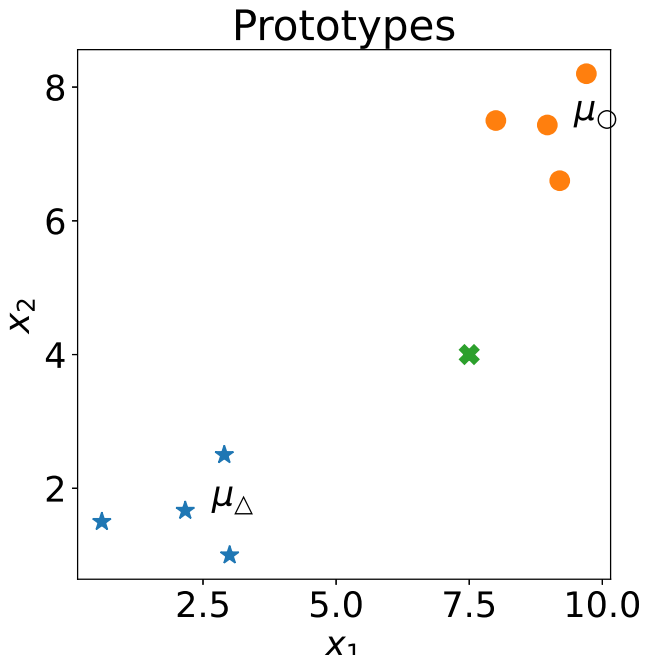


Figure 3.3: Mean values as prototypes for different categories are used to determine the label of a new data point

3.2.1 Inference

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

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

Algorithm 1: **N**earest **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 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: $\vec{x} \in \mathbb{R}^D, \vec{\mu}_k, k \in \{1, \ldots, K\}$
Result: means $\vec{\mu}_k, k \in \{1 \ldots k\}$
`# Init means and counters for each class`
`# Computation of class means`
for *class k in K* **do**
  $\vec{\mu}_k \leftarrow \frac{1}{N_k} \sum_{i=1}^{N_k} \vec{x_i}$;
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

$$\vec{\mu}_k = \frac{1}{N_k} \sum_{i=1}^{N_k} \vec{x_i} \tag{3.6}$$

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

$$\vec{\mu}_k = \frac{1}{N_k} \sum_{i=1}^{N_k - 1} \vec{x_i} + \frac{1}{N_k} \vec{x_{N_k}} \tag{3.7}$$

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

$$\frac{1}{N_k} = \frac{1}{N_k - 1} \cdot \frac{N_k - 1}{N_k} \tag{3.8}$$

Now we can rewrite the equation as

$$\vec{\mu}_k = \frac{1}{N_k} \sum_{i=1}^{N_k - 1} \vec{x_i} + \frac{1}{N_k} \cdot \vec{x_{N_k}} = \frac{N_k - 1}{N_k} \frac{1}{N_k - 1} \sum_{i=1}^{N_k - 1} \vec{x_i} + \frac{1}{N_k} \cdot \vec{x_{N_k}} \tag{3.9}$$

If we compare the middle term $\frac{1}{N_k} \sum_{i=1}^{N_k - 1} \vec{x_i}$ with the original equation (3.1) we can see that it is just the mean of the previous iteration $\vec{\mu}_{k-1}$

$$\Rightarrow \vec{\mu}_k = \frac{N_k - 1}{N_k} \vec{\mu}_{k-1} + \frac{1}{N_k} \cdot \vec{x_k} \tag{3.10}$$

This equation can be used to iteratively compute the mean of a category. We can start with $\vec{\mu}_0 = \vec{0}$ and then compute the mean of each sample by using the equation (3.10). An implementation of this approach is shown in Algorithm 3. As you can see for this iterative approach we only need to store all $\vec{\mu}_k$ and N_k in memory. This is a huge advantage over the batched approach, especially if we have a lot of data.

You can see a visualization of the two approaches in Figure A.1.

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

Algorithm 3: NCC Means (Streaming)

3.3 Limitations

Once we implemented one 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 3.4. This brings up a new group of questions, specifically about the limitations of the NCC or when should we use the NCC and when should we not use it.

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

1. NCC should only be used for uncorrelated data, i.e. x_1 and x_2 are independent/without any correlation. The background here is the prediction by the model, the line between the two colors in Figure 3.4 is called the *Decision Boundary* (DB). We see that we have not a single miss-classification in this example. But this doesn't apply to all cases. Occasionally we will witness miss-classifications even in simple examples. This is due to the fact that the NCC is a linear classifier and therefore can only separate linearly separable data. Whenever we have *outliers*, *mislabeled data* or *noise* in the input data, it is inevidable that the NCC will not be able to separate the data with full accuracy.

2. The NCC does not consider correlation when classifying. It only computes and copares mean values of the data.
   Compare the decision boundaries in Figure 3.4 and Figure 3.5. In Figure 3.5 we see that the decision boundary is not optimal, several data points of the blue class would be classified as orange and vice versa. If we compute only the means, we can not successfully separate the correlated data with a single DB.

3. This also applies to a problem with more than two classes. If we have more than two classes, we can not use a single DB to separate the data. We would need to compute multiple DBs to separate the data, as visualized in Figure 3.6.

3.4 Linear Classification

In the previous section we motivated the NCC by using a psychological model. We also saw that the NCC is good for uncorrelated data, data that is linear

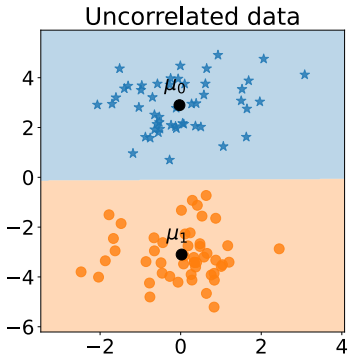


Figure 3.4: Decision Boundaries for uncorrelated data

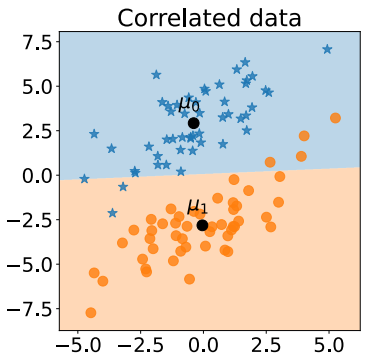


Figure 3.5: Decision Boundaries for correlated data

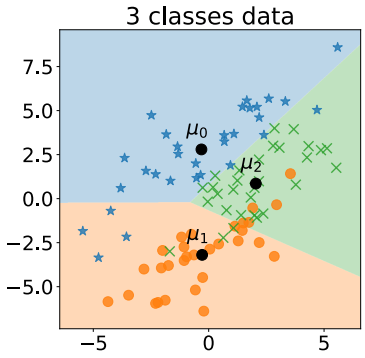


Figure 3.6: Decision Boundaries for 3 classes

separable.
Under the hood the NCC is computing Decision Boundaries to separate the data, this DB is a line in the feature space. This puts the NCC algorithm into the group of *Linear Classifiers*. Linear Classifiers are a group of algorithms that perform well on linearly separable data, just like the NCC. The NCC is a very simple linear classifier, but there are more complex ones. We will look into some of them in later chapters. But what is this line that separates the data? How can we compute it?
We will look into this now in a more mathematical way.

3.4.1 From Prototypes to Linear Classification

Now we will use the definition of NCC (3.5) to derive general linear classification. Let $\vec{x} \in \mathbb{R}^D$ be a data point and $\vec{\mu}_k \in \mathbb{R}^D$ be the prototype of class k. For two classes this would be $\vec{\mu}_0$ and $\vec{\mu}_1$.

Then we would find the class of $\vec{x}$ by computing the distance to each proto-

type and assigning the label of the prototype with the smallest distance.

$$k^* = \underset{k \in \{1,\dots,K\}}{\operatorname{argmin}} \; d(\vec{x}, \vec{\mu}_k) \tag{3.11}$$

or

$$k^* = \operatorname{argmin}(d(\vec{x}, \vec{\mu}_0), d(\vec{x}, \vec{\mu}_1)) \tag{3.12}$$
$$\Leftrightarrow d(\vec{x}, \vec{\mu}_0) > d(\vec{x}, \vec{\mu}_1) \tag{3.13}$$

This is the same as saying that $\vec{x}$ is closer to $\vec{\mu}_0$ than to $\vec{\mu}_1$. We can write this as

$$\Leftrightarrow \|\vec{x} - \vec{\mu}_0\|_2^2 > \|\vec{x} - \vec{\mu}_1\|_2^2 \tag{3.14}$$

where $\|\cdot\|_2$ is the Euclidean norm. We can square both sides of the inequality and get

$$\Rightarrow \|\vec{x} - \vec{\mu}_0\|_2 > \|\vec{x} - \vec{\mu}_1\|_2 \tag{3.15}$$

We can now expand the Euclidean norm to get

$$\Rightarrow \vec{x}^T\vec{x} - 2\vec{x}^T\vec{\mu}_0 + \vec{\mu}_0^T\vec{\mu}_0 > \vec{x}^T\vec{x} - 2\vec{x}^T\vec{\mu}_1 + \vec{\mu}_1^T\vec{\mu}_1 \tag{3.16}$$
$$\Leftrightarrow -2\vec{x}^T\vec{\mu}_0 + \vec{\mu}_0^T\vec{\mu}_0 > -2\vec{x}^T\vec{\mu}_1 + \vec{\mu}_1^T\vec{\mu}_1 \tag{3.17}$$
$$\Leftrightarrow \vec{\mu}_0^T\vec{x} - \frac{\vec{\mu}_0^T\vec{\mu}_0}{2} < \vec{\mu}_1^T\vec{x} - \frac{\vec{\mu}_1^T\vec{\mu}_1}{2} \tag{3.18}$$
$$\Leftrightarrow 0 < \underbrace{(\vec{\mu}_0 - \vec{\mu}_1)}_{\vec{\omega}}{}^T\vec{x} - \underbrace{\frac{1}{2}\left(\vec{\mu}_0^T\vec{\mu}_0 - \vec{\mu}_1^T\vec{\mu}_1\right)}_{\beta} \tag{3.19}$$

$\vec{\omega}$ is called the *weight vector*, for NCC this is the difference vector between both means. $(\vec{\mu}_0 - \vec{\mu}_1)^T\vec{x}$ is called the *activation* of the input $\vec{x}$. From the previous chapter 2 we know that this is essentially just projecting $\vec{x}$ onto the difference vector, which will result in a constant value. The constant value is then compared to the bias β and if it is greater than β the input is classified as class 0, otherwise as class 1. In other words

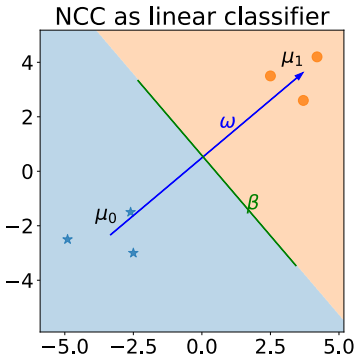
$$0 < \vec{\omega}^T\vec{x} + \beta \tag{3.20}$$

This is the general form of a linear classifier. Using this general form we can now compute the DB for the NCC example

$$\vec{\tilde{x}} = \left(\vec{\omega}^T \vec{x}\right) \tag{3.21}$$

$$\vec{\tilde{x}} - \beta = \begin{cases} > 0 & \text{class 0} \\ < 0 & \text{class 1} \end{cases} \tag{3.22}$$

with class 0 the orange class and class 1 the blue class.



NCC as linear classifier, the blue line visualizes the decision boundary $\vec{\omega}$ and the green line is the decision threshold β

This is more or less all we need to know from linear classifiers and we can now derive it from the NCC algorithm.

3.4.2 Linear Classification

We will now look a bit deeper into linear classification. Linear classification algorithms predict classes for given data points $\vec{x}$ by computing the activation of the input $\vec{x}$ and comparing it to a bias β

$$f(\vec{x}) = \vec{\omega}^T \vec{x} + \beta \tag{3.23}$$

where $\vec{\omega}$ is the decision boundary and β is the decision threshold. For two classes $\vec{\omega}$, the difference of the class means, is a vector and β is a scalar

$$\vec{\omega}_{\text{NCC}} = \vec{\mu}_0 - \vec{\mu}_1 \tag{3.24}$$

There are other ways to calculate $\vec{\omega}$ for other linear classifiers. Using this definition we can build the NCC as a linear classification model

$$f(\vec{x}) = \vec{\omega}_{\text{NCC}}^T \vec{x} + \beta_{\text{NCC}} \tag{3.25}$$

What does this geometrically mean?
First, we assume our data is sepearable by the diagonal through the 2nd and 4th quadrant. This is the same as saying that the data is linearly separable and we won't need a threshold β for now. $\vec{\omega}$ is parallel to $\vec{\mu}_0 - \vec{\mu}_1$ and therefore orthogonal to the DB. Then any point above the DB, perpendicular to $\vec{\omega}$, will be classified as class 0 and any point below the DB will be classified as class 1. You can see a visualization of that in Figure 3.4.2.

Linear Classifier without bias Linear Classifier with bias

Now, let's look at the bias value β. Figure 3.4.2 demonstrates that the bias is the value that is added to the activation of the input $\vec{x}$. You can literally say that we simply shift our DB up or down along the x_2-axis by β.

TODO: Add more details about linear classification, add linear classifier plots

Chapter 4

Nearest Neighbor Classifier

TODO:

Chapter 5

Hyperparameter Optimization

TODO:

Chapter 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].

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

$$z = \frac{x - \mu}{\sigma} \tag{6.1}$$

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

In Python we can implement this as follows:

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

Listing 6.1: z-Score in Python

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 6.1. 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

$$x_{scaled} = \frac{x - \min(\vec{x})}{\max(\vec{x}) - \min(\vec{x})} \tag{6.2}$$

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

The resulting variable is in the range $[0, 1]$ or any other. This method is also implemented in `sklearn.preprocessing.MinMaxScaler`. The implementation of the Min-Max-Scaling is very similar to the z-Score implementation in Code 6.1 and can be found in Code 6.2.

```
def min_max_scaling(X):
  x_min = X.min(axis=0)
  return (
    (X - x_min)
    / (X.max(axis=0) - x_min)
  )
```

Listing 6.2: Min-Max-Scaling in Python

Similar to the unnormlized data, the normalized data can be used in any ML model.

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

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 6.1: 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.

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.

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 words

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

Table 6.2: BoW-Feature Bi-Gram vector

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`.

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.

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 6.1.

In order to downweight the frequently occurring words, to be able to only select relevant words, we apply the inverse document frequency

$$\text{idf}(t, d) = \log \frac{|d|}{|d \text{ containing } t|} \tag{6.3}$$

This document frequency can be combined with the term frequency, the number of occurrences of a single word/token $|t|$ over the number of occurring words/-tokens $|d|$

$$\text{tf}(t, d) = \frac{|t|}{|d|} \tag{6.4}$$

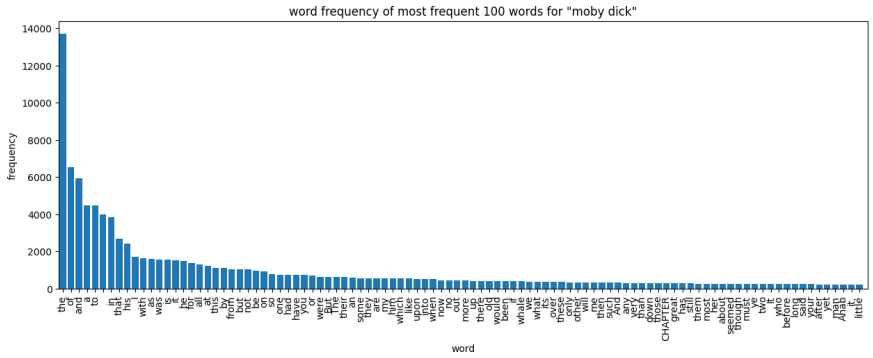


Figure 6.1: Word frequency distribution of 100 most frequent words in Moby Dick (https://www.gutenberg.org/ebooks/2701)

tf (6.4) and idf (6.3) can be combined by forming their product

$$\text{tfidf}(t, d) = \text{tf}(t, d) \cdot \text{idf}(t, d) \tag{6.5}$$

A model incorporating this 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` .

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.

Expensive N-Grams

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`.

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.

For those of you who don't know about hash-maps, you can think of it as a dictionary, where each word is a key and the value is the number of occurrences, you can read more about the inner workings of hash maps in the great 2020 blog post by Adam Gold [1].

Example #1

Let us look at an example of how to use these techniques in Python on a specific task. We will use the 20 newsgroups data set, which is a collection of 20,000 newsgroup documents, partitioned (nearly) evenly across all newsgroups. You can find more information about this data set here: `http://qwone.com/~jason/20Newsgroups/`. Essentially, we will train a model to classify the newsgroup of a given document. We will use the `sklearn.datasets.fetch_20newsgroups` function to download the data set and then use the `sklearn.feature_extraction.text.TfidfVectorizer` to transform the text into a BoW representation. After that we will train a NCC model and KNN model on the data set.

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.neighbors import NearestCentroid, KNeighborsClassifier

# Download the data set
newsgroups_train = fetch_20newsgroups(subset='train')
newsgroups_test = fetch_20newsgroups(subset='test')

# Transform the text into a BoW representation
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(newsgroups_train.data)
X_test = vectorizer.transform(newsgroups_test.data)

# Train a Nearest Centroid Classifier
clf = NearestCentroid()
clf.fit(X_train, newsgroups_train.target)
acc = (clf.predict(X_test) == imdb_test.target).mean()
print(f"Accuracy: {acc}")

# Train a KNN Classifier
clf = KNeighborsClassifier()
clf.fit(X_train, newsgroups_train.target)
acc = (clf.predict(X_test) == imdb_test.target).mean()
```

```
print(f"Accuracy: {acc}")
```

Listing 6.3: 20 newsgroups example

Model	Accuracy
Nearest Centroid Classifier	**0.692113648433351**
K-Nearest Neighbors Classifier	0.6591874668082847

Table 6.3: Accuracy of NCC and KNN on 20 newsgroups data set

Running the code in Code 6.3 will result in the accuracy scores shown in Table 6.3. You will notice that albeit the simplicity of the NCC model it performs better than the KNN model. Furthermore, the NCC model is also substantially faster to train.

A better approach for this problem would be to use the Logistic Regression (LogReg) model. We will look into this model in the future (Chapter 12), for now we will consider it a black box and apply it to the news group data set to demonstrate its performance on this task.

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression

# Download the data set
newsgroups_train = fetch_20newsgroups(subset='train')
newsgroups_test = fetch_20newsgroups(subset='test')

# Transform the text into a BoW representation
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(newsgroups_train.data)
X_test = vectorizer.transform(newsgroups_test.data)

# Train a Logistic Regression Classifier
clf = LogisticRegression(random_state=0)
clf.fit(X_train, newsgroups_train.target)
print(f"Accuracy: {clf.score(X_test, newsgroups_test.target)}")
```

Listing 6.4: 20 newsgroups example with Logistic Regression

Model	Accuracy
Nearest Centroid Classifier	0.692113648433351
K-Nearest Neighbors Classifier	0.6591874668082847
LogisticRegression	**0.8274030801911842**

Table 6.4: Accuracy of NCC, KNN and LogReg on 20 newsgroups data set

After running the code in Code 6.4 we can see that the LogReg model performs significantly better than the NCC and KNN models (see Table 6.4). This is due to the fact that the LogReg model is able to learn non-linear decision boundaries, which the NCC and KNN models are not able to do, because they are linear classifiers. We will look into LogReg in the future, but for now it is important to understand that the NCC model is a linear classifier and therefore can only learn linear decision boundaries. And apparently, the data set is not linearly separable, which is why the linear classification models perform so poorly.

Example #2

In this example we will look into a different data set, the *IMDB* data set [2] (`http://ai.stanford.edu/~amaas/data/sentiment/`). This data set contains 50,000 movie reviews from the Internet Movie Database, labeled by sentiment (positive/negative/unsupervised). We will use the `sklearn.datasets.load_files` function to download the data set and then use the `sklearn.feature_extraction.text.TfidfVectorizer` to transform the text into a BoW representation. Similar to the previous example, we will classify the reviews using a NCC model and a LinReg model.

```
from sklearn.datasets import load_files
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.neighbors import NearestCentroid
from sklearn.linear_model import LogisticRegression

# load the data set
imdb_train = load_files('aclImdb/train')
imdb_test = load_files('aclImdb/test')

# Transform the text into a BoW representation
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(imdb_train.data)
X_test = vectorizer.transform(imdb_test.data)

# Train a Nearest Centroid Classifier
clf = NearestCentroid()
clf.fit(X_train, imdb_train.target)
print(f"Accuracy: {clf.score(X_test, imdb_test.target)}")

# Train a Logistic Regression Classifier
clf = LogisticRegression(random_state=42, solver='newton-cg', C
    =100.)
clf.fit(X_train, imdb_train.target)
acc = (clf.predict(X_test) == imdb_test.target).mean()
print(f"Accuracy: {acc}")
```

Listing 6.5: IMDB example

As the results in Table 6.5 show, the NCC model performs better on this task than the LogReg model. But an accuracy of 62.31% is not very good, especially

Model	Accuracy
Nearest Centroid Classifier	**0.62312**
Logistic Regression	0.19984

Table 6.5: Accuracy of NCC and LogReg on IMDB data set

considering that a random guess would result in an accuracy of 33.3%. One option would be to reduce the problem to a binary classification problem, i.e. positive or negative sentiment. Doing so would result in an accuracy of 62.89% for the NCC model and 17.56% for the LogReg model. As you can see, the NCC model still performs better than the LogReg model, but still not very good, frankly the improvement was rather sobering.

To improve on the $\approx$ 60% accuracy we could use a different model architecture, e.g. a neural network. In future chapters we will look closer into the architecture powering so called Multi-Layer Perceptrons (MLPs). But for now

it is important to understand that MLPs are able to learn non-linear decision boundaries, which is why they are able to perform better on this task.

Implementing an MLP is as easy as replacing the NCC model with a MLP model from the `sklearn.neural_network.MLPClassifier` class.

```
from sklearn.datasets import load_files
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.neural_network import MLPClassifier

# load the data set
imdb_train = load_files('aclImdb/train')
imdb_test = load_files('aclImdb/test')

# Transform the text into a BoW representation
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(imdb_train.data)
X_test = vectorizer.transform(imdb_test.data)

# Train a MLP Classifier
clf = MLPClassifier(
  hidden_layer_sizes=(100, 100),
  max_iter=10,
  alpha=1e-4,
  solver='sgd',
  verbose=10,
  random_state=42,
  learning_rate_init=.1
)
clf.fit(X_train, imdb_train.target)
acc = (clf.predict(X_test) == imdb_test.target).mean()
print(f"Accuracy: {acc}")
```

Listing 6.6: IMDB example with MLP

The results in Table 6.6 show that the MLP model performs significantly better than the NCC and LogReg models.

Model	**Accuracy**
Nearest Centroid Classifier	0.62312
Logistic Regression	0.19984
MLP Classifier	**0.849**

Table 6.6: Accuracy of NCC, LogReg and MLP on IMDB data set

Great, we were able to improve the accuracy from $\approx 60\%$ to $\approx 85\%$.

But this can't be it. We can do better than that. And we will.

A more advanced technique is to use word embeddings, which we will look into now.

6.4.2 Word Embeddings

The following section describes techniques which apply neural networks to generate word embeddings. For now, we will look into how to use these embeddings as feature extractors. Later, in Chapters 18 and 19, we will look into how to train these embeddings models and how to use them for other tasks. Word embeddings are a more advanced technique to represent words as vectors as the previosuly introduced OHE approach of BoW.

Over the past two years word embeddings have enjoyed great popularity and research among the ML community. Especially with the rise of Large Language Models (LLMs) like BERT [3] and GPT-3 [4], the introduction of ChatGPT [5] in late 2022 and the groundbreaking Open Source community around architectures of 2023 like LLaMa [6], LLaMa2 [7] and Mixtral [8], these vectors became more and more relevant.

They are a dense representation of words, which means that they do not contain many 0 values. This is in contrast to the sparse representation of BoW-Features, which contain many 0 values. Word embeddings are usually trained on a large corpus of text, e.g. Wikipedia, and then used as a feature extractor for other tasks. The most popular word embedding is the *Word2Vec* embedding, which is trained on a large corpus of text coming from Wikipedia. Word Embeddings are quite versatile, they can be used for many different tasks, e.g. word similarity, word analogies, text classification, etc. We will look into the word similarity and word analogy tasks in a future chapter. The quintessence of Word Embeddings is that they are able to encode semantic and syntactic information of words. The most prominent example to demonstrate this encoded information is the following.

Let the embedding vector for the word "King" be $\vec{a} = \begin{pmatrix}1 & 1 & 0\end{pmatrix}$, the embedding vector of the word "Man" $\vec{b} = \begin{pmatrix}1 & 0 & 0\end{pmatrix}$ and the embedding vector for the word "Woman" be $\vec{c} = \begin{pmatrix}0 & 0 & 1\end{pmatrix}$. Then we can compute the embedding vector for the word "Queen" $\vec{d}$ by subtracting "Man" from "King" and adding "Woman" to create $\vec{d} = a - b + c = \begin{pmatrix}0 & 1 & 1\end{pmatrix}$.

This is of course a great simplification of the actual process. Embedding vectors have way more than just 3 dimensions and will most likely not contain beautiful integers as in this example.

We will look in Chapter 13 into a technique called Principal Component Analysis (PCA), which can be used to reduce the dimensionality of embedding vectors to 2 or 3 dimensions. This allows us to visualize these high dimensional vectors.

You can check out `https://projector.tensorflow.org/` for a great visualization of word embeddings.

Word2Vec

Word2Vec is a word embedding technique that was introduced in 2013 by Mikolov et al. [9]. The idea behind Word2Vec is to train a neural network to predict the context of a word. The context of a word is defined as the words surrounding the word in a given text. The neural network is trained on a large corpus of text, e.g. Wikipedia, and then used as a feature extractor for other tasks. A library that implements Word2Vec is `gensim` (`https://radimrehurek.com/gensim/`). Using Word2Vec is as easy as downloading a pre-trained model and then using it as a feature extractor.

```
import gensim.downloader as api

# Download the pre-trained model
model = api.load("word2vec-google-news-300")

# Get the embedding vector for the word "King"
```

```
print(model["king"])
```

Listing 6.7: Word2Vec example

The code in Code 6.7 will download the pre-trained Word2Vec model (≈1.7 GB) from Google and then print the embedding vector for the word "King".

As you can see, we can generate embeddings for single words, but also for sentences and even paragraphs.Albeit the flexibility of the Word2Vec model, the model is not capable of generating embeddings for unknown words, i.e. words that are not in the original training vocabulary, e.g. "Kinging" and "Queening". This is a problem that is solved by the FastText model, which we will look into in the next section.

FastText

FastText is a word embedding technique that was introduced in 2016 by Bojanowski et al. [10]. The idea behind FastText is to train a neural network to predict the context of a word, similar to Word2Vec. But instead of using words as the smallest unit, FastText uses character N-Grams. This allows FastText to generate embeddings for unknown words, i.e. words that are not in the original training vocabulary, e.g. "Kinging" and "Queening". Gensim also provides a FastText model, which can be used in the same way as the Word2Vec model.

```
import gensim.downloader as api

# Download the pre-trained model
model = api.load("fasttext-wiki-news-subwords-300")

# Get the embedding vector for the word "King"
print(model["king"])
```

Listing 6.8: FastText example

The code in Code 6.8 will download the pre-trained FastText model (≈1 GB) from Wikipedia and then print the embedding vector for the word "King".

GloVe

Another word embedding technique is GloVe, which was introduced in 2014 by Pennington et al. [**?**]. Other than FastText and Word2Vec, GloVe is not a neural network based model, but a matrix factorization technique. The idea behind GloVe is to factorize the word-word co-occurrence matrix. The word-word co-occurrence matrix is a matrix that contains the number of times a word i appears in the context of word j. The GloVe model is also available in Gensim.

```
import gensim.downloader as api

# Download the pre-trained model
model = api.load("glove-wiki-gigaword-300")

# Get the embedding vector for the word "King"
print(model["king"])
```

Listing 6.9: GloVe example

The code in Code 6.9 will download the pre-trained GloVe model (≈1.4 GB) from Wikipedia and then print the embedding vector for the word "King".

Comparison

Now that we have looked into three different word embedding techniques, let us compare them. We will use the same example as in the previous section, the 20 newsgroups data set. We will use the `sklearn.datasets.fetch_20newsgroups` function to download the data set and then use the embedding models to transform the text into a vector representation. After that we will train a NCC model and MLP model on the data set.

TODO: Code sample not working, fix it

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.neighbors import NearestCentroid
from sklearn.neural_network import MLPClassifier
import gensim.downloader as api

# Download the data set
newsgroups_train = fetch_20newsgroups(subset='train')
newsgroups_test = fetch_20newsgroups(subset='test')

# Download the pre-trained models
word2vec_model = api.load("word2vec-google-news-300")
fasttext_model = api.load("fasttext-wiki-news-subwords-300")
glove_model = api.load("glove-wiki-gigaword-300")

# Transform the text into a vector representation
X_train_word2vec = [word2vec_model[x] for x in newsgroups_train.
    data]
X_test_word2vec = [word2vec_model[x] for x in newsgroups_test.data]
# Train a Nearest Centroid Classifier
clf = NearestCentroid()
clf.fit(X_train_word2vec, newsgroups_train.target)

# Train a MLP classifier
clf = MLPClassifier(
  max_iter=10,
  verbose=10,
  random_state=42,
)
clf.fit(X_train_word2vec, newsgroups_train.target)

print("Word2Vec")
print(f"NCC Accuracy: {clf.score(X_test_word2vec, newsgroups_test.
    target)}")
print(f"MLP Accuracy: {(clf.predict(X_test_word2vec) ==
    newsgroups_test.target).mean()}")

del X_train_word2vec, X_test_word2vec

X_train_fasttext = [fasttext_model[x] for x in newsgroups_train.
    data]
X_test_fasttext = [fasttext_model[x] for x in newsgroups_test.data]

# Train a Nearest Centroid Classifier
clf = NearestCentroid()
clf.fit(X_train_fasttext, newsgroups_train.target)

# Train a MLP classifier
clf = MLPClassifier(
  max_iter=10,
```

```
  verbose=10,
  random_state=42,
)
clf.fit(X_train_fasttext, newsgroups_train.target)

print("FastText")
print(f"NCC Accuracy: {clf.score(X_test_fasttext, newsgroups_test.
    target)}")
print(f"MLP Accuracy: {(clf.predict(X_test_fasttext) ==
    newsgroups_test.target).mean()}")
del X_train_fasttext, X_test_fasttext

X_train_glove = [glove_model[x] for x in newsgroups_train.data]
X_test_glove = [glove_model[x] for x in newsgroups_test.data]

# Train a Nearest Centroid Classifier
clf = NearestCentroid()
clf.fit(X_train_glove, newsgroups_train.target)

# Train a MLP Classifier
clf = MLPClassifier(
  max_iter=10,
  verbose=10,
  random_state=42,
)
clf.fit(X_train_glove, newsgroups_train.target)

print("GloVe")
print(f"NCC Accuracy: {clf.score(X_test_glove, newsgroups_test.
    target)}")
print(f"MLP Accuracy: {(clf.predict(X_test_glove) ==
    newsgroups_test.target).mean()}")
```

Listing 6.10: 20 newsgroups example with embeddings

The code in Code 6.10 will download the pre-trained embedding models and then transform the text into a vector representation. After that we train a NCC model and MLP model on the data set.

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 Classic 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 6.2.

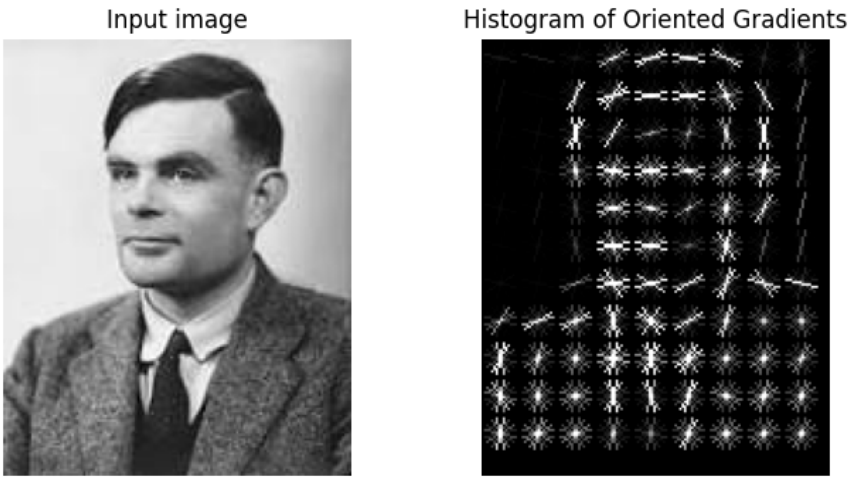


Figure 6.2: HOG feature extraction

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 [11], 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 [12] or Chapter 18.

Essentially, CNNs are combinations of convolutional filter masks that are stacked on top of each other adding more and more complexity to the model. The first layers learn simple features like edges and the last layers learn more complex features like shapes and objects. The last layers are then used as features for our ML model. Originally, when using the ImageNet data set, the models are trained on the image classification task. But we can use the features of the last layers for any other task, e.g. object detection, because the features are still very meaningful and valid. Instead of using the full pretrained model, we can also use only the first layers of the model and train the last layers on our own data set, or we simply only use the first few pretrained layers as feature extractors.

You can see the representation of different channels from a specific layer in Figure 6.3 as well as the combination of all channels in Figure 6.4.

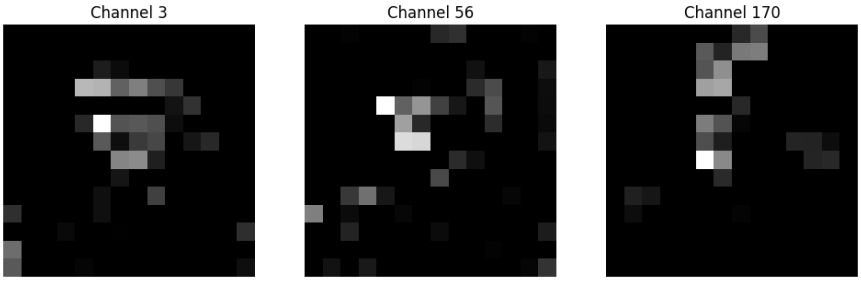


Figure 6.3: CNN feature extraction of different channels

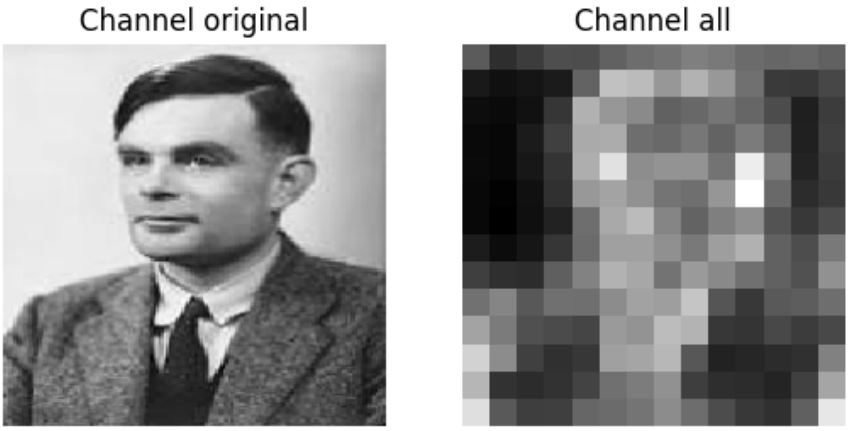


Figure 6.4: CNN feature channels combined

You can imagine image channels as a encoding of spacial frequencies, similar to the HOG features. But other than HOG features, CNNs are able to learn these features in a general way by themselves, which is why they are so powerful.

Consider this, color images have three channels, red, green and blue. Each of these channels encodes a different spacial frequency. Let $X \in \mathbb{R}^{H \times W \times 3}$ be an image with height H, width W and three channels and random content (see Figure 6.5). Then we can define three matrices $T_{red}, T_{green}, T_{blue} \in \mathbb{R}^{H \times W \times 3}$ that select the corresponding channel of each pixel.

$$\begin{pmatrix} \{\vec{x}_{1,1,1}, \vec{x}_{1,1,2}, \vec{x}_{1,1,3}\} & \{\vec{x}_{1,2,1}, \vec{x}_{1,2,2}, \vec{x}_{1,2,3}\} & \dots & \{\vec{x}_{1,W,1}, \vec{x}_{1,W,2}, \vec{x}_{1,W,3}\} \\ \{\vec{x}_{2,1,1}, \vec{x}_{2,1,2}, \vec{x}_{2,1,3}\} & \{\vec{x}_{2,2,1}, \vec{x}_{2,2,2}, \vec{x}_{2,2,3}\} & \dots & \{\vec{x}_{2,W,1}, \vec{x}_{2,W,2}, \vec{x}_{2,W,3}\} \\ \vdots & \vdots & \ddots & \vdots \\ \{\vec{x}_{H,1,1}, \vec{x}_{H,1,2}, \vec{x}_{H,1,3}\} & \{\vec{x}_{H,2,1}, \vec{x}_{H,2,2}, \vec{x}_{H,2,3}\} & \dots & \{\vec{x}_{H,W,1}, \vec{x}_{H,W,2}, \vec{x}_{H,W,3}\} \end{pmatrix} \tag{6.6}$$

Then we can extract the three channels as follows. For the red channel we select

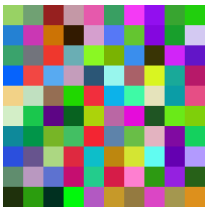


Figure 6.5: Random image

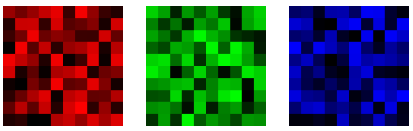


Figure 6.6: Image channels individually

the first channel of each pixel

$$T_{red} = \begin{pmatrix} \{1,0,0\} & \{1,0,0\} & \dots & \{1,0,0\} \\ \{1,0,0\} & \{1,0,0\} & \dots & \{1,0,0\} \\ \vdots & \vdots & \ddots & \vdots \\ \{1,0,0\} & \{1,0,0\} & \dots & \{1,0,0\} \end{pmatrix} \tag{6.7}$$

For the green channel we select the second channel of each pixel

$$T_{green} = \begin{pmatrix} \{0,1,0\} & \{0,1,0\} & \dots & \{0,1,0\} \\ \{0,1,0\} & \{0,1,0\} & \dots & \{0,1,0\} \\ \vdots & \vdots & \ddots & \vdots \\ \{0,1,0\} & \{0,1,0\} & \dots & \{0,1,0\} \end{pmatrix} \tag{6.8}$$

And for the blue channel we select the third channel of each pixel

$$T_{blue} = \begin{pmatrix} \{0,0,1\} & \{0,0,1\} & \dots & \{0,0,1\} \\ \{0,0,1\} & \{0,0,1\} & \dots & \{0,0,1\} \\ \vdots & \vdots & \ddots & \vdots \\ \{0,0,1\} & \{0,0,1\} & \dots & \{0,0,1\} \end{pmatrix} \tag{6.9}$$

Applying those three matrices to our image X we get three images $X_{red}, X_{green}, X_{blue} \in \mathbb{R}^{H \times W \times 3}$ as individually shown in Figure 6.6.

TODO:

Implement example for CNN feature extraction

Chapter 7

Machine Learning Pipelines

Recall the pipeline diagram from the first chapter.

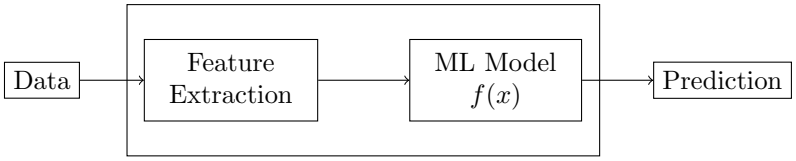


Figure 7.1: Machine Learning Pipeline

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

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 [13] for this is implemented by the guys from scikit-learn [14].

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

The basic concept of the scikit-learn API are so called *Estimators*, which define a overall interface for all sorts of algorithms, like classifiers or feature extractors. Each estimator implements a `fit` method, which takes a data set as input and learns from it. A `transform` method that applies the estimator to data which can yield classification predictions or transformed data. And optionally a `set_params` and a `get_params`-method to set and get configuration of the estimator.

Example
Imagine we want to detect *spam* in emails. Doing so we would end up with a pipeline similar to the one in figure 7.2.

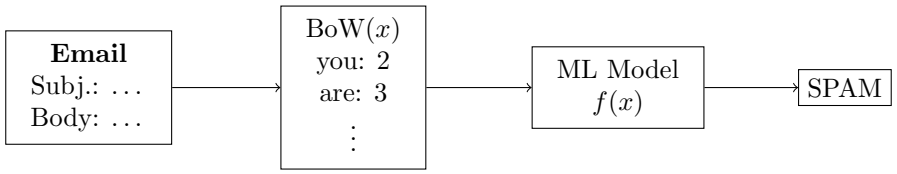


Figure 7.2: Machine Learning Pipeline for Spam Detection

7.3 Pipelines

To build a full *Pipeline* object, we can chain Estimators to run sequentially. This allows combining Feature Extractors and Classifiers into a full Pipeline object. Once chained, the Pipeline works just like an Estimator itself. It has a `fit` and a `transform` method. The `fit` method will call the `fit` method of each estimator in the pipeline sequentially. A second big advantage of such Pipeline object is that all parameters of it can be optimized jointly. This means that we can optimize the parameters of the feature extractor and the classifier at the same time. This Pipeline approach is great and allows us to quickly train and run full ML systems/architectures.
15-20 years ago, people needed to code all of this from scratch, in C. So nowadays we must appreciate this great work and use it to our advantage.
Recently we see a new spring of manual implementations of ML algorithms, especially in C and C++. With the publication of OpenAI's Whisper [15] or the leak of the weights of Meta's first generation of LLaMa [**?**] and the release of the second generation of LLaMa [**?**] we see a lot of people trying to implement and hack ML models from scratch, such as

- llama2.cpp by Andrej Karpathy
- llama.cpp by Georgi Gerganov
- whisper.cpp by Georgi Gerganov

TODO:

Chapter 8

Metrics

TODO:

Chapter 9

Model Evaluation

TODO:

Chapter 10

Perceptron

Perceptrons are among KNN and NCC one of the simplest, yet powerful and popular algorithms. They are the easiest form of Neural Networks that we know of and are a great introduction into the field of Artificial Neural Networks.

But first we make a small recap of the NCC algorithm. Remember, we defined the prototypes corresponding to each class as the means

$$\vec{\mu}_{\triangle} = \frac{1}{N_{\triangle}} \sum_{\vec{x} \in \mathcal{X}_{\triangle}} \vec{x} \tag{10.1}$$

$$\vec{\mu}_{\circ} = \frac{1}{N_{\circ}} \sum_{\vec{x} \in \mathcal{X}_{\circ}} \vec{x} \tag{10.2}$$

to classify a new data point $\vec{x}$ we saw that we must compute the distance to each class mean. For the example in Figure 3.3 this translates to

$$\mathrm{d}(\vec{x}, \vec{\mu}_{\triangle}) > \mathrm{d}(\vec{x}, \vec{\mu}_{\circ}) \tag{10.3}$$

After several steps of rearranging the definitions of both sides, we ended up with

$$0 > (\vec{\mu}_{\circ} - \vec{\mu}_{\triangle})^T \vec{x} + \frac{1}{2} \left(\vec{\mu}_{\triangle}^T \vec{\mu}_{\triangle} - \vec{\mu}_{\circ}^T \vec{\mu}_{\circ} \right) \tag{10.4}$$

That can be transformed into the general form of a linear classifier

$$\vec{w}^T \vec{x} + \beta = \begin{cases} > 0 \text{ if } \vec{x} \text{ belongs to class } \triangle \\ < 0 \text{ if } \vec{x} \text{ belongs to class } \circ \end{cases} \tag{10.5}$$

For NCC we defined $\vec{w}$ to be the difference vector of the class means $(\vec{\mu}_{\circ} - \vec{\mu}_{\circ})$ and β some constant offset. This class of linear classifiers is super important in Machine Learning, especially if we want to perform classifications fast and efficiently. A lot of technologies that must perform fast classifications are using those linear classification algorithms, something like a perceptron.

We will briefly recall the visual representation of linear classifiers as showcased in Figure 3.4.2. For a new data point $\vec{x}$ we can assign a class label by projecting $\vec{x}$ onto $\vec{\omega}$ via $\vec{\omega}^T \vec{x}$. If this resulting scalar is bigger than the threshold β, we assign the class label $\triangle$, otherwise $\circ$. Now, if we would classify a lot of samples we might see a distribution showcased in Figure 10.1 for $\circ$ like the grey line and for $\triangle$ like the orange one.

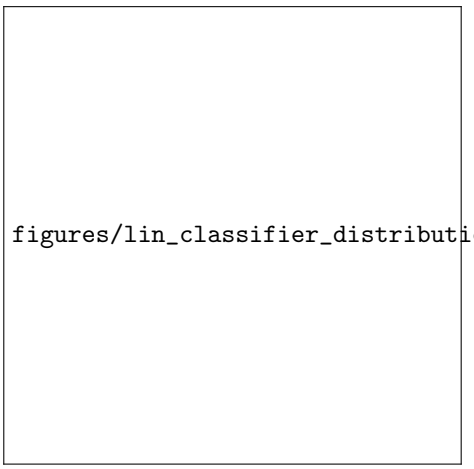


Figure 10.1: Distribution of the class labels $\triangle$ and $\circ$ for a linear classifier.

In the previous chapter we saw that this β threshold would be "perfect", but we might want to optimize for a different metric, for instance precision or recall, which would adjust the threshold.

Good, now that we refreshed our knowledge on linear classifiers, we can move on to the perceptron.

We know linear classifiers predict classes and what they are. But we didn't answer yet how to calculate the parameters $\vec{\omega}$ and β in a general way, how can we find this vector efficiently?

From the NCC algorithm we remember there's a very simple way to compute $\vec{\omega}$, namely the difference vector of the class means. And we learned in the NCC chapter that this method restricts the algorithm vastly and has several drawbacks.

Usually, the computation of $\vec{\omega}$ is done a little different for general linear classificators. Instead of coming up with a definition to compute $\vec{\omega}$, we must define a so called *error/loss function*.

10.1 Error Functions

Error functions are functions of the group $(\vec{x}, \vec{y}, \vec{\omega})$. Given data points $\vec{x} \in \mathbb{R}^d$ and their corresponding class labels $\vec{y} \in \mathbb{R}^{d_y}$ and a vector of parameters $\vec{\omega} \in \mathbb{R}^d$ it computes the error of our model with the given weights $\vec{\omega}$ on the data points $\vec{x}$.

For now and the rest of this chapter we will assume to only have binary labels, e.g. $\vec{y} \in \{-1, 1\}$. This is not a big restriction, since we can always transform any multi-class problem into a binary one by using the one-vs-all approach that we will see at the end of this chapter (Section 10.9).

There are two very popular error functions that we will discuss in this chapter, namely the *Square Error* (Adaline Loss) and the *Perceptron Loss*.

The table 10.1 shows the two error functions that we will discuss in this chapter. The first one is the square error, also known as Adaline loss. The second one is the perceptron loss.

Error Function	Definition
Square Error	$(\vec{x}, \vec{y}, \vec{\omega}) = \frac{1}{2}\left(\vec{y} - \vec{\omega}^T \vec{x}\right)^2$
Perceptron Loss	$(\vec{x}, \vec{y}, \vec{\omega}) = \max\left(0, -\vec{y}\vec{\omega}^T \vec{x}\right)$

Table 10.1: Error functions for linear classifiers.

10.2 Rosenblatt's Perceptron

The second error function was invented by Frank Rosenblatt (Figure **??**) in 1957. He was the inventor of the Perceptron algorithm and is therefore the creator of the field of ANNs.

> He studied perceptrons, so that the fundamental laws of organization which are common to all information handling systems, machines and men included, may eventually be understood
>
> - quite a bold statement.

We will now briefly introduce ANNs, but we will go into more detail later, in Chapter 18.

10.2.1 Artificial Neural Networks

Rosenblatt was influenced in his ANN architecture design by human brains. His implementation is a very, very simplified computational model that uses some aspects of their biological role model. An ANN consists of *input neuros* $(x_1, x_2, \dots)$ that are weighted with corresponding *synaptic weights* $(w_1, w_2, \dots)$. The sum of these weighted inputs is then used to compute the prediction. The method $f(...)$ is a non-linear function, that we omit for now, which will decide upon the final label

$$f(\vec{x}) = \begin{cases} 1 \text{ if } \vec{x} \text{ is preferred stimulus} \\ -1 \text{ otherwise} \end{cases} \tag{10.6}$$

You can see a visualization of such Perceptron in Figure 10.3.

This is pretty abstract now, but apart from the non-linear function $f(...)$, this is exactly what we saw in the previous chapter.

When Rosenblatt implemented his Perceptron it required a full room of hardware, with massive mechanical relays and giant memory rigs. Nowadays, we can run the perceptron algorithm on a microcontroller, or even on a small chip that is capable of matrix-vector multiplication. The original Perceptron algorithm was used to classify handwritten text, we will use exactly the same use case to showcase the algorithm.

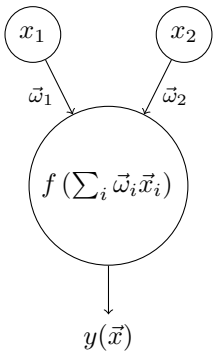


Figure 10.3: Visualization of a Perceptron.

10.3 The Perceptron Learning Algorithm

Now we will look deeper into the Perceptron algorithm. The goal is to perform binary classification of multivariate data points $\vec{x} \in \mathbb{R}^d$.

The input for the algorithm are N tuples of data points $\vec{x}_i$ and their corresponding class labels y_i, where $y_i \in \{-1, 1\}$, such that

$$y_n = \{1 \text{ if } \vec{x}_n \text{ belongs to class } \quad -1 \text{ if } \vec{x}_n \text{ does not belong to class} \tag{10.7}$$

additionally the algorithm receives a parameter called *learning rate* η, we will define this in a second.

The output of the algorithm is a vector of weights $\vec{\omega}$ and a bias β that define the linear classifier

$$\vec{\omega}^T \vec{x} + \beta = \begin{cases} \geq 0 \text{ if } y_n = +1 \\ < 0 \text{ if } y_n = -1 \end{cases} \tag{10.8}$$

Recall the optimization note in the NCC lecture, where we learned that we can also write β into $\vec{\omega}$ by adding a constant dimension to $\vec{x}$

$$\vec{\hat{\omega}} = [\beta, \vec{\omega}]^T, \vec{\hat{x}} = [1, \vec{x}]^T \tag{10.9}$$

The Perceptron Error Function This is a reformulation of the perceptron loss function from Table 10.1

$$(\vec{x}, y, \vec{\omega}) = \max\left(0, -y\vec{\omega}^T \vec{x}\right) = -\sum_{m \in \mathcal{M}} \vec{\omega}^T \vec{x}_m y_m \tag{10.10}$$

where $\mathcal{M}$ is the set of all misclassified data points.

10.3.1 Classification Error as Function of weights

You can see a plot of this error function in Figure **??**. On the left side we have all correct classified labels, on the right the misclassified ones.

Now, let's look at a real 2D example data set. On the right side you can see a plot for the error, if we project our $\vec{\omega}$ vector onto any point in our coordinate system on the left side.

So each point on the right grid represents a potential $\vec{\omega}$ vector. The color of the point represents the error of this $\vec{\omega}$ vector. To find the best value for $\vec{\omega}$ we must find the minimum of this error function. But usually we can't just simply create this plot, nor can we try out every possible value for $\vec{\omega}$ in finite time. So in general, what we do in ML is, to randomly initialize $\vec{\omega}$ and evaluate our error function. But instead of evaluating the regular error function, we evaluate it's gradient. This gives us the steepness of the error function in $\vec{\omega}$. To minimize the error through $\vec{\omega}$ we then adjust $\vec{\omega}$ in the opposite direction of the gradient. This works, because the gradient of the error function points into the direction of the biggest error. This process is called *Gradient Descent (GD)* and is a very popular optimization technique in ML.

10.4 Gradient Descent

Gradient Descent is, as we just saw, an algorithm that can optimize a function $f(\vec{x}, \vec{\omega})$ by iteratively adjusting the parameters $\vec{\omega}$ in the opposite direction of the gradient of f. Hence we can use GD to minimize the error function of the Perceptron algorithm.

We randomly initialize $\vec{\omega}$ and then iteratively update it by subtracting the gradient of the error function $(\vec{x}, y, \vec{\omega})$. And this is how GD works in detail We have an old value for $\vec{\omega}$, $\vec{\omega}^{\text{old}}$, e.g. randomly initialized, compute the gradient of the error function using $\vec{\omega}^{\text{old}}$ by summing over all training samples that is scaled by η, the learning rate, to compute

$$\vec{\omega}^{\text{new}} = \vec{\omega}^{\text{old}} - \eta \sum_{i=1}^{X} \nabla_{\vec{\omega}}(\vec{x}, y, \vec{\omega}) \tag{10.11}$$

Here you can see why this algorithm is called GD, we subtract the current gradient from our current weights. The learning rate η determines how fast we descent. The resulting $\vec{\omega}^{\text{new}}$ will be the new adjusted weights from our model. We can now repeat this process until we reach a certain threshold in the error, or until we reach a certain number of iterations.

10.4.1 Stochastic Gradient Descent

10.4.2 Mini-Batch Gradient Descent

10.5 Perceptron Training

10.6 Problems with the Perceptron Algorithm

10.7 Application Example: Handwritten Digits

10.8 Derivation of the Perceptron Error Function

10.9 Combining multiple Perceptrons

10.9.1 One-vs-All

10.9.2 One-vs-One

10.9.3 Application Example: Handwritten Digits (multi-class)

TODO:

Chapter 11

Decision Trees

Decision trees are another group of supervised learning algorithms. They are used for both classification and regression problems. Decision trees are easy to understand and interpret, and they are very useful for exploratory data analysis. They are also the basis for more sophisticated methods we will introduce at the end of this chapter. Similar to the KNN algorithm, decision trees are also non-parametric methods, which use the data as the model itself. But different to the KNN algorithm, decision trees are non linear algorithms in the classical sense. We will also see that there is a relation of decision trees to linear models.

11.1 Classification Trees

In this chapter and in the book we will only look into classification trees. The regression trees are very similar, but we will not discuss them here.

Before we begin, we must introduce important concepts that construct decision trees.

A *tree* is a hierarchical structure consisting of *nodes* and *edges*. Nodes are connected by edges, and the edges are directed from the *parent* node to the *child* node. For simplicity, we will only consider binary trees, where each node has at most two children. Nodes without children are called *leaves* or *terminal nodes*, and nodes with children are called *internal nodes* or *decision nodes*. The top node of a tree is called the *root node*. You can see a very simple example of a tree in Figure 11.1.

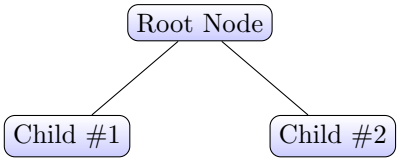


Figure 11.1: Simple decision tree.

11.1.1 Motivational Examples

Let us consider a simple example to motivate the idea of decision trees. Imagine you want to plan a dinner party. And you need to decide whether you host the party inside or outside. You have a lot of friends, and you want to invite as many as possible.

You come up with the decision tree shown in Figure 11.2. From looking at

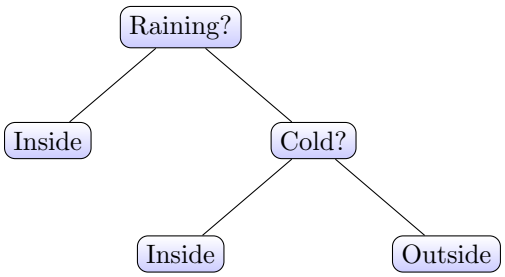


Figure 11.2: Decision tree for planning a dinner party. Left nodes answer the short questions with *yes*, right nodes with *no*.

this tree we can make two major observations. First, the tree is a sequence of binary questions that we can answer with *yes* or *no*. Second, the tree is a sequence of decisions that lead to a final decision. We can also see that the tree is a sequence of *if-then-else* statements.
If it is raining, we host the party inside, else we ask the next question.
If it is cold, we host the party inside, else we host the party outside.

This sounds simple to implement, but how do we come up with these questions?

11.1.2 Building a Decision Tree

In a very simplistic way to build a decision tree, we need to follow the following three steps

1. Split the data in "the best way possible"
2. continue this process with every new left and right side, until satisfied
3. create leaf nodes for final splits, assign label of majority of remaining samples to the leaf node

But what does "the best way possible" mean? We will try to understand this based on the following example.

11.1.3 Linearly Seperable Data

Given the following linear seperable data X and accomodating labels y, find the correct split to perform binary classification

$$X = \begin{bmatrix} 0.3 \\ 0.37 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{11.1}$$

The solution to this is rather obvious

$$f(x) = \begin{cases} 0 \text{ if } x \leq 0.3 \\ 1 \text{ if } x > 0.3 \end{cases} \tag{11.2}$$

or as a decision tree (Figure 11.3).

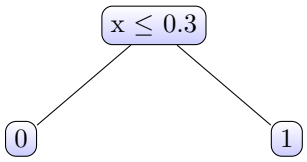


Figure 11.3: Decision tree for very simplistic linear seperable data.

We can break down the decision process for the numerical value x and selected threshold 0.3 into two steps

1. Select the best possible feature

   In this case, we only have one feature, so we do not need to select one

2. Find the value in x that separates the classes best

   In this case, we can see that the value 0.3 is the only value available that represents the class 0 and 0.37 the only sample of class 1

With increasing amount of data points this process becomes increasingly hard to perform, manually.

Consider the following data

$$X = \big(0.35, 0.6, 0.67, 0.8\big)^T \tag{11.3}$$

$$y = \big(0, 0, 1, 1\big)^T \tag{11.4}$$

The optimal solution is still very obvious

$$f(x) = \begin{cases} 0 \text{ if } x \leq 0.6 \\ 1 \text{ if } x > 0.6 \end{cases} \tag{11.5}$$

or as tree

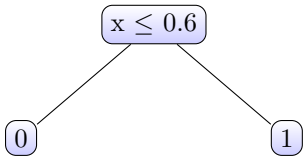


Figure 11.4: Decision tree for more linear seperable data.

11.1.4 Non-Linearly Seperable Data

Given the following non-linear seperable data X and accomodating labels y, find the correct split to perform binary classification

$$X = \big(0.3, 0.1, 0.21, 0.35, 0.6, 0.67, 0.8, 0.786, 0.97\big)^T \tag{11.6}$$

$$y = \big(0, 0, 1, 0, 1, 1, 0, 1, 0\big)^T \tag{11.7}$$

In this example we can not simply split the data by briefly looking at it and visually recognize the seperation. We need to find a way to split the data in a way that we can seperate the classes as good as possible. For this we need to understand what a split actually is, how we can evaluate the quality of a split and how we can find the best split. Additionally, when we have multivariate data, we need to understand how we can select the best feature to split on.

11.2 Information Gain

The information gain Gain tells us how much information we gain over x while looking at y. This metric can be used to evaluate the quality of a split. It measures the reduction of an impurity metric in a splitted data set. The information gain is defined as

$$\text{Gain}(S, V) = I(S) - \sum_{S_v \in V} \frac{|S_v|}{|S|} I(S_v) \tag{11.8}$$

with V a set of splits out of S. For two splits

$$\text{Gain}(S, V) = I(S) - \left(\frac{|S_1|}{|S|} I(S_1) + \frac{|S_2|}{|S|} I(S_2) \right) \tag{11.9}$$

11.3 Impurity Metrics

The impurity metric I is a metric that measures the impurity of a data set. The following metrics are calculated at the node level. The lower their value, the purer the observed data.

11.3.1 Entropy

The entropy is a measure of the uncertainty of a random variable and ranges from 0 to 1. It is defined as

$$I(S) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{11.10}$$

with p_i the probability randomly selecting a sample of class i out of the k classes in S.

In Python, we can calculate the entropy as follows

```
def entropy(s):
  counts = np.bincount(np.array(s, dtype=np.int64))
  percentages = counts / len(s)
```

```
  return -np.sum([
    pct*np.log2(pct)
    for pct in percentages
    if pct > 0
  ])
```

11.3.2 Gini Impurity

The Gini impurity is a measure of the probability of a random sample being classified incorrectly if it was randomly labeled according to the distribution of labels in the subset. Hence it combines the probability of randomly selecting an item i

$$p_i = \frac{|S_i|}{|S|} \tag{11.11}$$

with the probability of misclassifying an item i

$$\sum_{j \neq i} p_j = 1 - p_i = \frac{|S| - |S_i|}{|S|} \tag{11.12}$$

Thereby, the Gini impurity is defined as

$$I(S) = \sum_{i=1}^{c} p_i(1 - p_i) = \sum_{i=1}^{c}(p_i - p_i^2) \tag{11.13}$$

$$= \underbrace{\sum_{i=1}^{c} p_i}_{:=1} + \sum_{i=1}^{c} p_i^2 = 1 - \sum_{i=1}^{c} p_i^2 \tag{11.14}$$

In Python, we can calculate the Gini impurity as follows

```
def gini_impurity(s):
    counts = np.bincount(np.array(s, dtype=int))
    percentages = counts / len(s)
    return 1 - (percentages**2).sum()
```

Both metrics are very similar, but the Gini impurity is slightly faster to compute due to the lack of logarithm. They also share some properties. A low impurity measure translates to a low likelihood of misclassification. A high value translates to a high likelihood of misclassification.

11.3.3 Prediction Error

Another metric is the prediction error. It is defined as

$$I(S) = 1 - \max_i p_i \tag{11.15}$$

and is essentially the inverse of the maximum probability of correctly classifying a sample for any given class in S.
This metric is not as useful as the other two to construct decision trees, but it is useful to evaluate the quality of a tree to optimize it. One way of optimizing decision trees is to reduce their amount of decision nodes. This can be done by pruning the tree. Pruning is the process of removing decision nodes from a tree. We will discuss this in more detail shortly.

Implemented in Python, the prediction error looks like this

```
def prediction_error(s):
    counts = np.bincount(np.array(s, dtype=int))
    percentages = counts / len(s)
    return 1 - np.max(percentages)
```

11.3.4 Comparison of Impurity Metrics

For comparison we will compute the impurity meytics for 101 different combinations of binary labels for 100 samples.

In Python, we could do something like this

```
g = list(); e = list(); e2 = list(); err = list(); bins = list()
for i in range(101):
    values = [0] * (100 - i) + [1] * i
    g.append(gini_impurity(values))
    e.append(entropy(values))
    e2.append(np.array(entropy(values))/2.)
    err.append(error(values))
    bins.append(values)
```

and we can visualize these results with the following code

```
import matplotlib.pyplot as plt

def draw_bins(bins, alpha=0.5):
  for idx, val in enumerate(bins):
    unique, counts = np.unique(val, return_counts=True)
    if idx == 0:
      counts = np.array([len(val), 0])
    elif idx == len(bins) - 1:
      counts = np.array([0, len(val)])
    plt.bar(
      [idx,idx+0.5],
      height=counts/len(val),
      width=0.5,
      color=['b', 'g'],
      label=['class 0', 'class 1'],
      alpha=alpha
    )
```

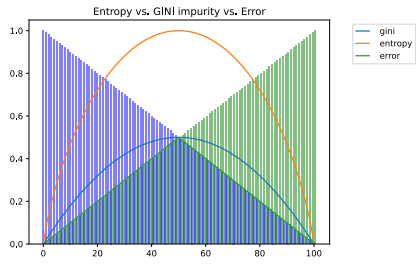


Figure 11.5: Impurity metrics for 101 different combinations of binary labels for 100 samples.

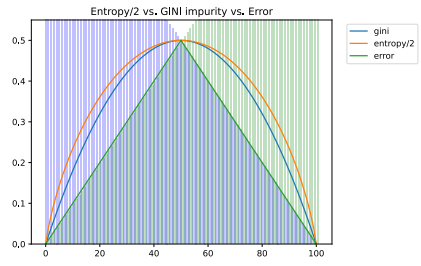


Figure 11.6: Rescaled impurity metrics for 101 different combinations of binary labels for 100 samples.

The entropy ranges from 0 pure to 1 impure, whereas the Gini impurity ranges from 0 pure to 0.5 impure as you can see in Figure 11.5. To visualize the

relationship between the two metrics, we can rescale the entropy by dividing it by 0.5. This allows us to see that the Gini impurity lies between the Entropy and the prediction error and is not a differently scaled Entropy (see Figure **??**).

Depending on the chosen metric the resulting trees can vary. Sometimes this makes a small impact, sometimes a bigger one. Overall, Entropy and Gini impurity are implemented in many algorithms. CART (Classification and Regression Trees) uses the Gini impurity, whereas ID3 (Iterative Dichotomiser 3) uses the Entropy.

Most of these implementations like C5 are highly optimized using methods like *boosting* that improve the structure of the trees by selecting better splits.

11.4 Disadvantages of Decision Trees

The biggest and main disadvantage of decision trees is that deep trees are prone to overfitting the training data. On the other hand, shallow trees increase the risk of biased predictions.

One solution for this is called *Random Forrest*. It is an ensemble method that combines multiple decision trees to reduce the risk of overfitting without sacrificing bias. Random Forrests are more robust and generally better solvers than single decision trees.

11.5 Decision Trees in scikit-learn

You can find an implementation based on NumPy in the accomodating Jupyter notebook to this chapter. In this section we will look at the implementation of decision trees in scikit-learn and apply it to the Iris data set.

The decision tree classifier is implemented in the `DecisionTreeClassifier` class. It is, as it name suggests, the implementation of the classification variant of Decision Trees and has the following parameters

- `criterion`: The impurity metric to use. Either `gini` or `entropy`.
- `max_depth`: The maximum depth of the tree. If `None`, the tree is grown until all leaves are pure.
- `min_samples_split`: The minimum number of samples required to split an internal node.
- `min_samples_leaf`: The minimum number of samples required to be at a leaf node.
- `min_impurity_decrease`: A node will be split if this split induces a decrease of the impurity greater than or equal to this value.
- `class_weight`: Weights associated with classes in the form {`class_label: weight`}.
- `random_state`: The seed of the pseudo random number generator to use when shuffling the data.
- `max_features`: The number of features to consider when looking for the best split.

You can see an example usage of the decision tree classifier in Listing 11.1.

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split

# load data
iris = load_iris()
X = iris.data

# split into train and test set
X_train, X_test, y_train, y_test = train_test_split(
  X, iris.target, test_size=0.2, random_state=42
)

dtc = DecisionTreeClassifier()
dtc.fit(X_train, y_train)

print('Accuracy:', dtc.score(X_test, y_test))
```

Listing 11.1: Example usage of the decision tree classifier.

As discussed in the HPO chapter (Chapter 5), we can use for instance the `GridSearchCV` class to find the best parameters for our model. You can see an example usage of this in Listing 11.2.

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, GridSearchCV

# load data
iris = load_iris()
X = iris.data

# split into train and test set
X_train, X_test, y_train, y_test = train_test_split(
  X, iris.target, test_size=0.2, random_state=42
)

dtc = DecisionTreeClassifier()

params = {
  'criterion': ['gini', 'entropy'],
  'max_depth': [None, 2, 4, 6, 8, 10],
  'min_samples_split': [2, 4, 6, 8, 10],
  'min_samples_leaf': [1, 2, 3, 4, 5],
  'min_impurity_decrease': [0.0, 0.1, 0.2, 0.3, 0.4, 0.5],
  'class_weight': [None, 'balanced'],
  'random_state': [42],
  'max_features': [None, 'auto', 'sqrt', 'log2']
}

grid = GridSearchCV(dtc, params, cv=5, n_jobs=-1)
grid.fit(X_train, y_train)

print('Best score:', grid.best_score_)
print('Best params:', grid.best_params_)
print('Accuracy:', grid.score(X_test, y_test))
```

Listing 11.2: Example usage of the decision tree classifier with grid search.

TODO:

Finalize example

Chapter 12

Regression

TODO:

Chapter 13

Principal Component Analysis (PCA)

TODO:

Chapter 14

Linear Discriminant Analysis (LDA)

TODO:

Chapter 15

Support Vector Machines (SVM)

TODO:

Chapter 16

Naive Bayes

TODO:

Chapter 17

Clustering

TODO:

Chapter 18

Artificial Neural Networks

TODO:

Chapter 19

Deep Learning

TODO:

Chapter 20

Natural Language Processing (NLP)

TODO:

Chapter 21

Computer Vision

TODO:

Chapter 22

Generative Artificial Intelligence

TODO:

Chapter 23

Reinforcement Learning

TODO:

Appendix A

Visualizations

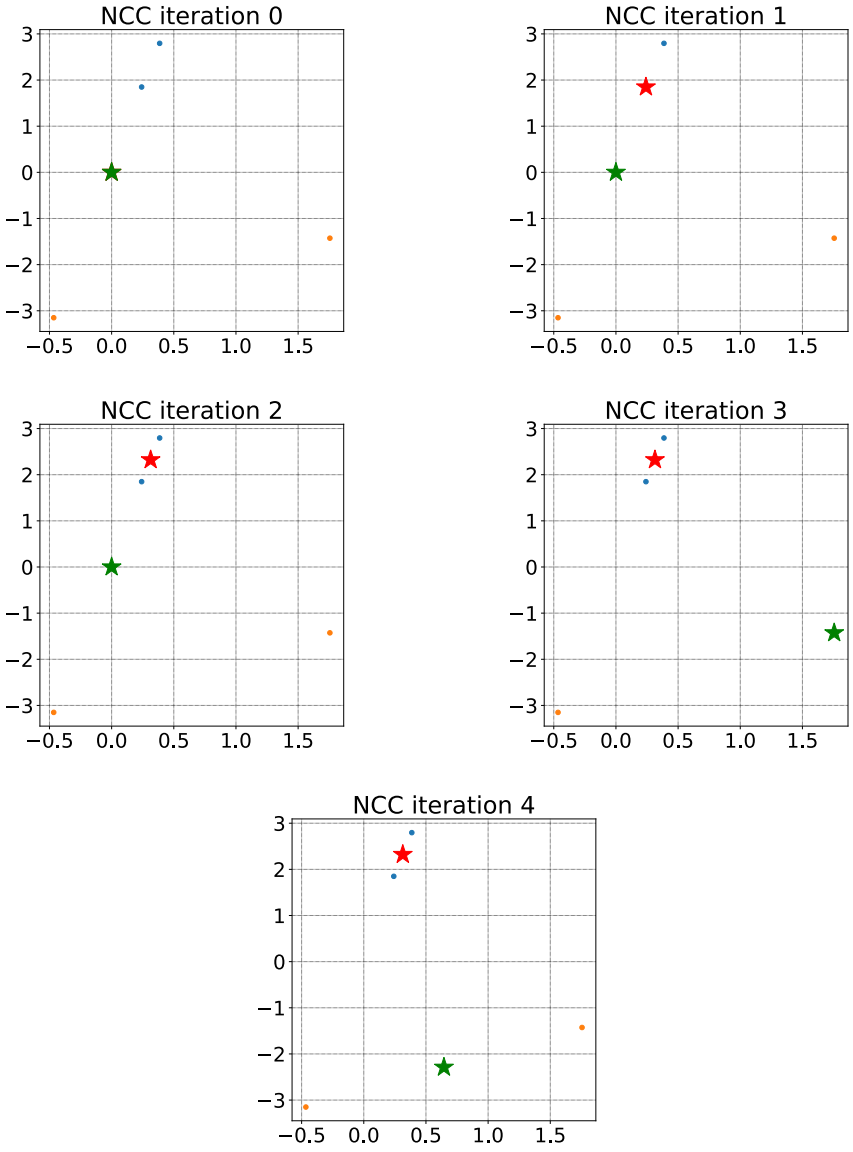


Figure A.1: Nearest Centroid Classifier (NCC) with streaming updates for 4 samples of two classes. The blue class is represented by the red mean and the orange class by the green mean. The blue samples are represented by blue dots and the orange samples by orange dots. The mean of the blue class is updated after adding the first (Figure A) and second blue sample (Figure A). The mean of the orange class is updated after adding the first (Figure A) and last orange sample (Figure A).

Bibliography

[1] A. Gold, "Python hash tables under the hood," 2020, "[Online; accessed 2023-12-29]". [Online]. Available: https://adamgold.github.io/posts/python-hash-tables-under-the-hood/

[2] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. [Online]. Available: http://www.aclweb.org/anthology/P11-1015

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[4] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[5] OpenAI, "Introducing chatgpt," 2022, "[Online; accessed 2023-12-30]". [Online]. Available: https://openai.com/blog/chatgpt

[6] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[7] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.

[8] Mistral.AI, "Mixtral of experts: A high quality sparse mixture-of-experts." 2023, "[Online; accessed 2023-12-30]". [Online]. Available: https://mistral.ai/news/mixtral-of-experts/

[9] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[10] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *arXiv preprint arXiv:1607.04606*, 2016.

[11] 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*. Ieee, 2009, pp. 248–255.

[12] P. Zettl, "Machine learning methods for localization and classification of insects in images," 2022, "[Online; accessed 2023-12-24]". [Online]. Available: https://github.com/philsupertramp/inet

[13] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[15] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," 2022. [Online]. Available: https://arxiv.org/abs/2212.04356