ML-Toolbox

Table of Contents

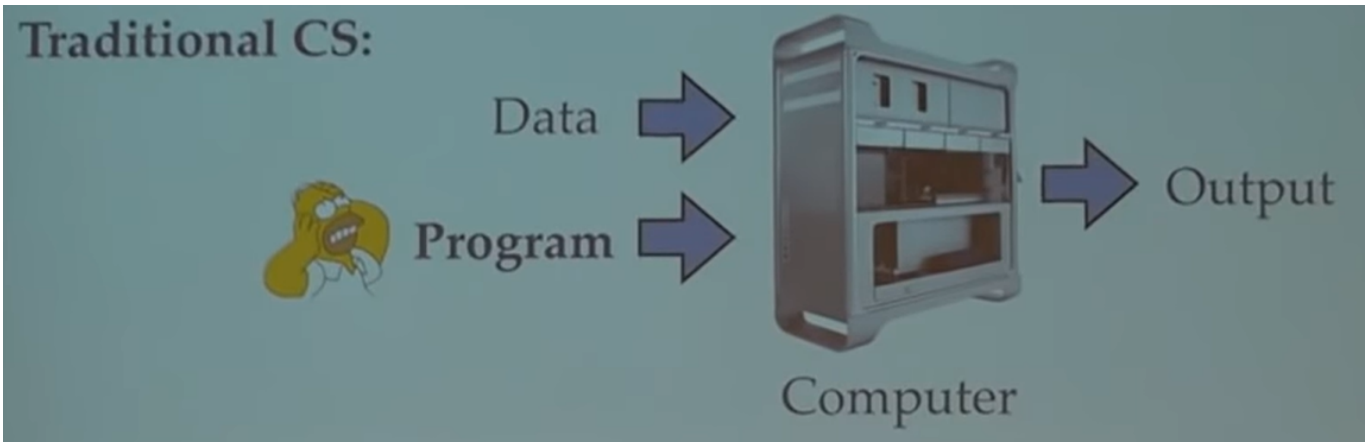
About

Every machine learning algorithm comes with its own set of assumptions about the data. The goal of the ML Toolbox is to understand these assumptions, and make an informed decision when selecting the best model for a given problem. By aligning the right algorithm with the characteristics of the data, we can optimize performance and achieve better results.

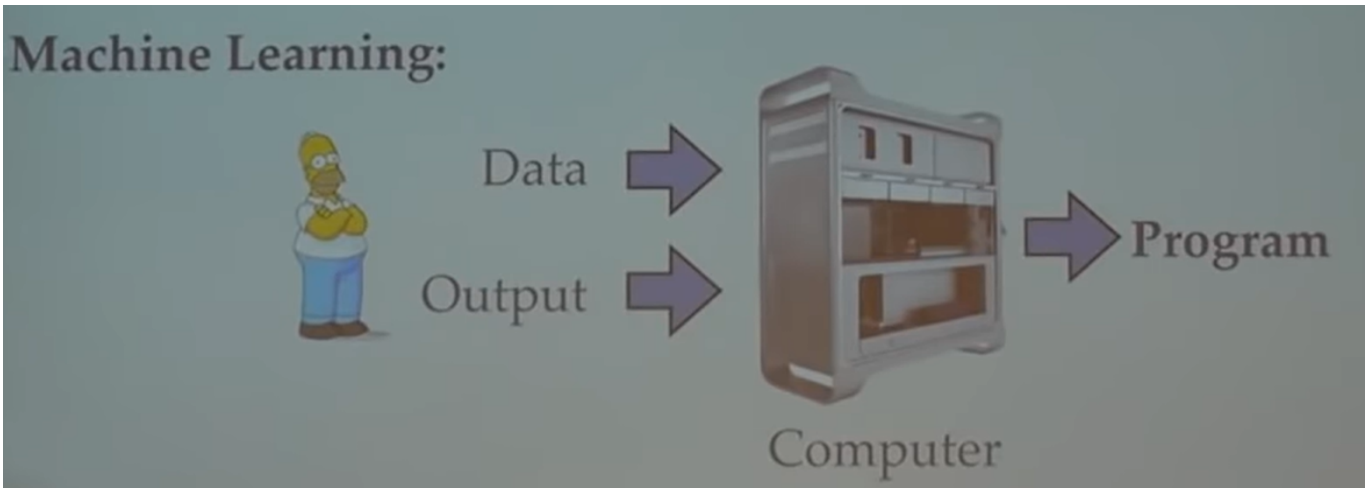
Introduction to Machine Learning

Traditional Programming vs Machine Learning

Traditional Programming is based on the idea of writing a program, giving it an input and getting an output. This works well for all the tasks where the rules can be clearly defined. Consider the problem of classifying a number as odd or even. This can be done by a simple if-else program.



For problems where the rules can not be clearly defined, we use Machine Learning to generate these rules for us. Consider the problem of classifying an image as cat or dog. Writing a program for this would be very difficult. Machine Learning is the idea where we provide the computer with data and corresponding outputs and get the program. This phase is called training. Now we use these program, along with new data like traditional programming to get an output. This phase is called inference.



Core Idea behind Machine Learning

Machine Learning is a subset of Artificial Intelligence (AI). While AI aims to imitate human thinking, Machine Learning focuses on using statistics to uncover patterns in data. For instance, in games like chess, AI uses strategies like minimax, similar to how humans strategize, while Machine Learning methods such as Linear Regression aim to draw the best-fitting line through data points, relying on statistics and pattern recognition rather than mimicking human thought processes.

At the heart of machine learning is the quest to find a function `f(x)` that closely approximates the relationship between inputs and outputs in the real world. Unlike traditional programming, where functions are manually defined, machine learning algorithms learn from data to automatically derive the most suitable function or model for a given task.

Introduction to ML-Toolbox

The ML-Toolbox is like a toolkit full of different machine learning methods, each offering its own form of `f(x)`. The trick is picking the right one for the job, which is kind of like choosing a setting on a tool – it depends on what we are trying to do. Neural networks are popular, but they're just one tool in the box, giving us outputs in the form of weights and biases.

The core concept behind the ML-Toolbox is to grasp the diverse range of algorithms capable of generating forms of `f(x)`. Some widely used algorithms include Decision Trees, Neural Networks, Support Vector Machines, Random Forests, and K-Nearest Neighbors. The goal isn't to say which method is the best. Instead, it's about knowing when each method works well and when it might struggle. It's like knowing when to use a screwdriver versus a hammer.

File Structure

```
ML-Toolbox
├ assets
│ ├ data                                  // datasets
│ │ ├ articles.csv
│ │ ├ gender.csv
│ │ ├ modified_mumbai_house_price.csv
│ │ ├ mumbai_house_price.csv
│ │ ├ student_marksheet.csv
│ │ ├ titanic.csv
│ │ ├ un_voting.csv
│ ├ img
│ ├ scripts
│ ├ notes
├ Concept Learning
│ ├ titanic survival prediction.ipynb
├ K Nearest Neighbors
│ ├ gender prediction.ipynb
│ ├ house price prediction.ipynb
│ ├ article recommendation.ipynb
├ Perceptron
│ ├ gender prediction.ipynb
├ Naive Bayes
├ Logistic Regression
│ ├ gender prediction.ipynb
├ Linear Regression
│ ├ house price prediction.ipynb
├ Support Vector Machine
│ ├ gender prediction.ipynb
├ Kernels
│ ├ Perceptron
│ │ ├ gender prediction.ipynb
│ ├ Linear Regression
│ │ ├ house price prediction.ipynb
│ ├ Support Vector Machine
│ ├ gender prediction.ipynb
├ Neural Networks
│ ├ gender prediction.ipynb
│ ├ house price prediction.ipynb
├ K Means Clustering
│ ├ grouping students.ipynb
│ ├ article recommendation.ipynb
├ README.md
```

Results

Pending Section

The following sections are still in progress:

- Kernels
- Neural Networks
- Decision Trees
- Association Rule Mining
- KD Trees
- Gaussian Processes
- Notes

References

- Big thanks to Prof. Kilian Weinberger for the Cornell CS4780 course, Machine Learning for Intelligent Systems. Majority of the content in this repository is inspired from the lectures.
- MIT 6.036 Machine Learning by Prof. Tamara Broderick.
- Bias Variance Tradeoff by MIT OpenCourseware and The Stanford NLP Group.
- Additional resources to understand kernelizations.
- Neural Networks and Deep Learning Online Book by Michael Nielsen.
- Blog on Curse of Dimensionality.
- Kaggle for providing several datasets used in this repository.

License

Concept Learning

Hypothesis Space

Formally, in the context of machine learning, the function `f(x)` represents a hypothesis `h` within a hypothesis space `H`. For instance, if we select a decision tree as our function `f(x)`, then the hypothesis space `H` would encompass the set of all possible decision trees. The objective is to find a hypothesis `h` that serves as the most accurate approximation of the true function `f`.

Introduction to Concept Learning

Imagine we want to learn to identify birds from a group of animals. We can start by looking at different birds and try to understand the features which make them different from other animals. We might look at a sparrow and derive that if the animal has wings, feathers, brown color, a short tail, etc. then it is a bird. However, by looking at other birds, we might realize that color has nothing to do with a bird, and we can come to a generalized idea that if it has feathers or wings, it is a bird.

Here, bird becomes a Concept, which we want to learn. Concept Learning is a boolean function defined over a set of all possible animals which returns true only if a given object is a member of a bird. The problem of inducing general functions from specific training examples is central to concept learning.

FIND-S Algorithm

Assumption

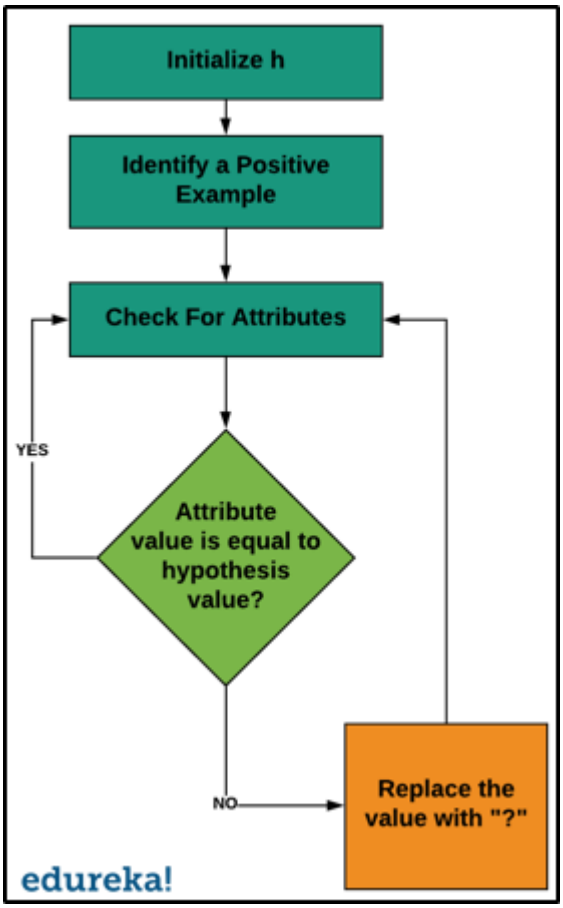
The FIND-S Algorithm works on a simple assumption: we can learn something by looking only at examples where it's true. For instance, if we want to find a thief, we ask people who have seen the thief what they look like (positive examples), but we don't bother asking those who haven't seen the thief (negative examples). By

gathering details from people who have seen the thief, we try to build a general picture of what the thief might look like.

When questioning witnesses, the algorithm starts with specific details and tries to find common traits. For example, one person might say the thief wore a black jacket and cap, while another might say they wore a shirt and cap. From these details, we can guess that the thief was at least wearing a cap. This process helps us build a broader understanding of the concept.

However, while FIND-S is a basic way to learn from examples, it's not very practical for real-life problems. Other methods are better at handling different situations and are more effective overall. Because of this, FIND-S hasn't been studied and improved as much as other, more advanced techniques.

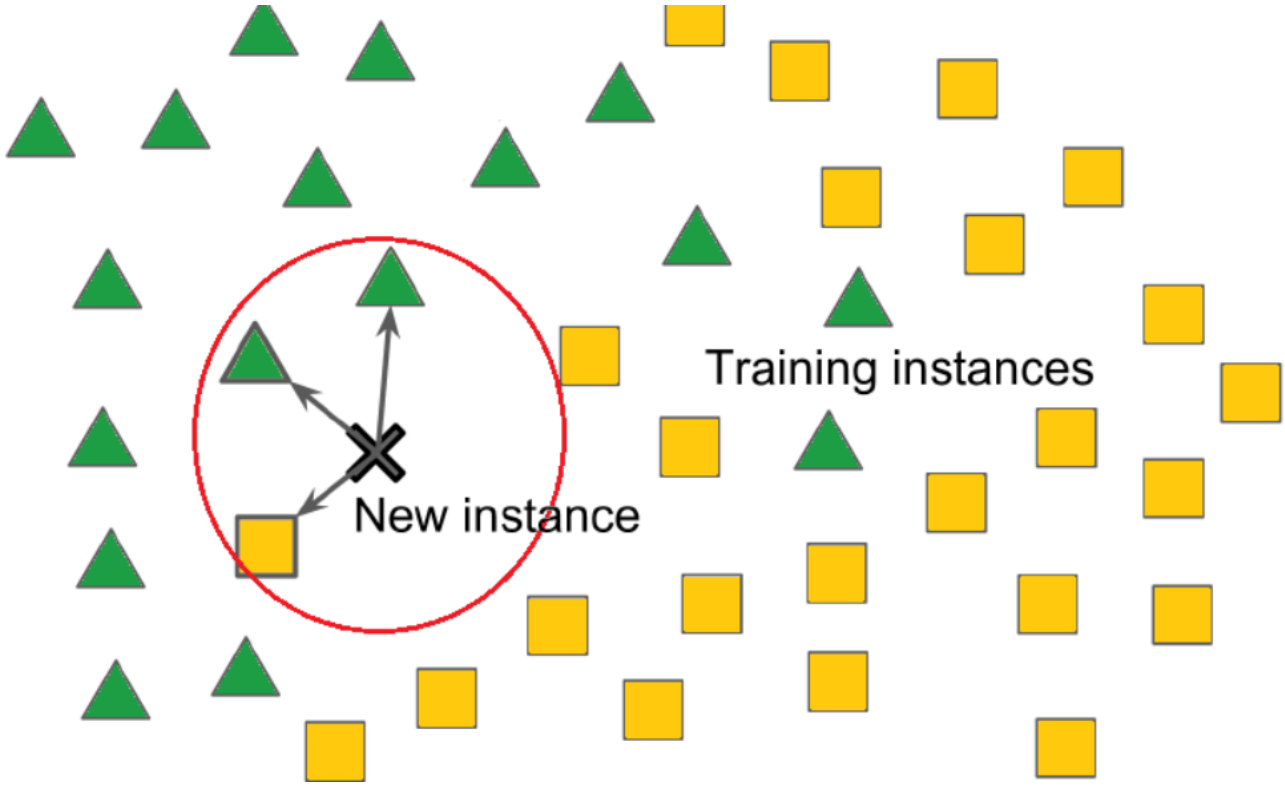
Algorithm



Results

Upon applying the FIND-S algorithm to the Titanic dataset, it yields a highly general hypothesis, suggesting limitations in its ability to understand the concept of survival in the Titanic data. However, if we consider only the first few instances, it gives us the following hypothesis: `['?', '?', '?', 'female', '?', '?', '?', '?', '?', '?']`. This finding suggests that gender emerges as a notable attribute among survivors, provided we consider only the first few instances.

K Nearest Neighbors (KNNs)



Introduction

K Nearest Neighbors (KNNs) is a supervised machine learning algorithm that can be used for both classification and regression tasks.

Assumptions

KNN operates under the assumption that instances that are close to each other in the feature space are likely to be similar. In other words, points that are similar to each other also tend to have similar target values. Thus the target value of a new instance is likely to be the same as its nearest neighbors.

However, this assumption may not always hold, especially in datasets where the relationship between features and target values is not strictly dependent on proximity. For instance, consider a dataset concerning customer preferences, where features such as age and income are considered. Customers with similar ages and incomes may still have vastly different preferences.

Algorithm

The KNN algorithm is only as good as its distance metric. The distance metric should be such that it captures the similarity between instances appropriately. For example, Euclidean Distance is a better metric for classifying handwritten digits based on pixel values, but it would prove to be a bad metric for calculating text similarity. Cosine Similarity would be a better metric for text similarity. Some commonly used distance metrics are:

1. Minkowski Distance: This is a generalized distance metric that includes Manhattan (p=1), Euclidean (p=2), and Chebyshev (p=infinity) as special cases. It is defined as:

Minkowski distance (*p*-norm): $D(x,x') = \sqrt[p]{\sum_d |x_d - x'_d|^p}$

The choice of distance metric depends on the amount of penalty one wants to assign to differences in each dimension. If p is lower, say 1, then the metric is less sensitive to outliers and treats each dimension equally. If p is higher, say infinity, then it is sensitive to outliers in any single dimension.

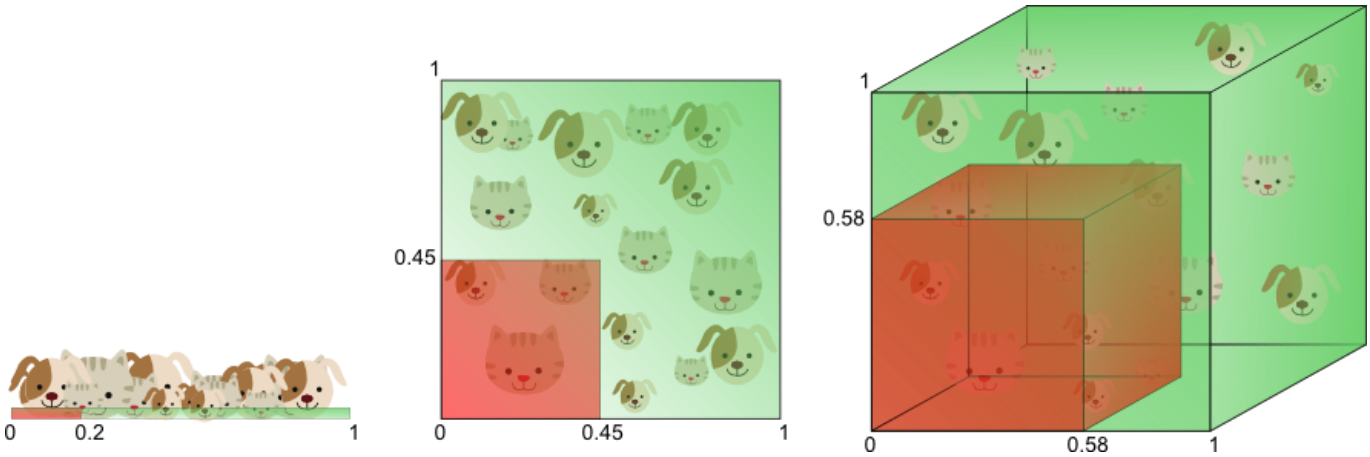
Consider a recommendation system with features such as number of pages read, time spent, etc. Here, if a user reads 100 pages vs 200 pages, it should not be penalized as heavily as spending 1 hour vs 10 hours. Manhattan distance (p=1) would be better. On the other hand, for classifying images where each pixel is equally important, Euclidean distance (p=2) would work better. Similarly, Chebyshev distance (p=infinity) can be used when one dimension is more important than others. For example, in medical diagnosis, a single severely abnormal symptom may be more significant than several marginally abnormal symptoms.

2. Cosine Similarity: Cosine similarity measures the cosine of the angle between two vectors. It is a measure of orientation and not magnitude. Cosine similarity is commonly used as a similarity metric for text data.

Once the similarity metric is defined, for any given test point, we look at its k nearest neighbors and take the majority vote of the classes of these neighbors for classification and average of the target values for regression.

The choice of the parameter k (the number of neighbors) is crucial, as it impacts the model's sensitivity to noise and generalization ability. A smaller k may result in a model that is sensitive to noise, while a larger k may lead to a model that is too generalized. The optimal k is often determined through validation methods.

Curse of Dimensionality in KNNs



K-Nearest Neighbors (KNN) is based on the assumption that data points close together in the feature space are more likely to belong to the same category. However, as the number of features (dimensions) increases, this assumption can break down due to the curse of dimensionality. In high-dimensional spaces with few data points (sparse data), identifying the true nearest neighbors becomes challenging.

One consequence of this challenge is that the nearest neighbor found by the algorithm might not truly be a neighbor in the meaningful sense. In reality, it could be far from the test point, appearing close only due to

the sparseness of the data. Consequently, the core assumption of KNN that nearby points are similar becomes meaningless in such scenarios.

In these cases, algorithms like the perceptron may be more suitable for classification tasks. The perceptron, for instance, can handle higher dimensions more gracefully and is less affected by the curse of dimensionality.

However, it's essential to note that there are instances where datasets possess large dimensions but low intrinsic dimensionality. In such cases, KNN can still be effective. For example, images often have high dimensions but low intrinsic dimensionality, meaning that important information can be captured in fewer dimensions. Techniques like Principal Component Analysis (PCA) can help in reducing the dimensions while preserving most of the important information, making KNN applicable even in high-dimensional scenarios.

Results

Classification

Our goal was to build a system that could classify names as belonging to boys or girls. We had two options:

- Full Name as Text: This keeps the name as it is, "John" or "Sarah".
- Encoded Name: We converted the name into a vector of 702 numbers. This vector was made up of the last letter and bigrams (like "ia" or "th").

We used various distance metrics such as Manhattan distance, Euclidean Distance, Cosine Similarity, Hamming Distance to measure the distance between vectors, but none of these methods worked well. This might be because we didn't have enough data to make sense of such a complex representation. This might be due to curse of dimensionality.

Since that didn't work, we decided comparing the names directly as text. We used minimum edit distance as the distance metric. This calculates the minimum number of changes (insertions, deletions, or replacements) needed to turn one name into another. This method of using minimum edit distance as distance metric proved to be more effective and achieved an accuracy of `82.45%` on test data. However this method has a shortcoming. Consider the names 'Prit' and 'Priti'. Clearly, the vowel on the end changes the gender. However, both of these names differ by edit distance of 1. Thus, we design a new metric for Indian names, accounting for the fact that addition of a vowel in end changes gender.

A special adjustment is made to edit distance when the edit distance is 1 and the names only differ by a vowel at the end (e.g., "Shrey" and "Shreya"). In such cases, based on domain knowledge, the labels of the training samples are swapped (using XOR on the label), which might represent handling specific domain nuances. This method provided a validation accuracy of `88.37%` and a test accuracy of `83.84%`.

Further, by using weighted KNN we get a validation accuracy of `87.60%` and a test accuracy of `87.69%`.

Regression

K-Nearest Neighbors (KNN) is often called "lazy learning" because it doesn't really build a model; it just remembers all the training data. When we look at the average errors over mumbai house price prediction task, we see that KNN has a score of `0.38 Cr`, but weighted KNN, where we use the inverse distances as weights, has a lower error of `0.30 Cr`.

This is quite lower error compared to linear regression. This is maybe because of features such as latitude and longitude. In places like Mumbai, where location strongly affects prices, linear regression falls short because it can't handle non-linear relationships like those between prices and coordinates. Also, by converting nominal and ordinal features such as age, type, and status to appropriate numeric values, we were able to improve accuracy. We use absolute distance as a metric for finding the nearest neighbour. This absolute distance represents median value and hence is more effective than linear regression which predicts the mean. While predicting quantities such as property prices or average salary, outliers can easily skew the mean but not the median. Hence median becomes a better estimate than mean in this case.

Retrival

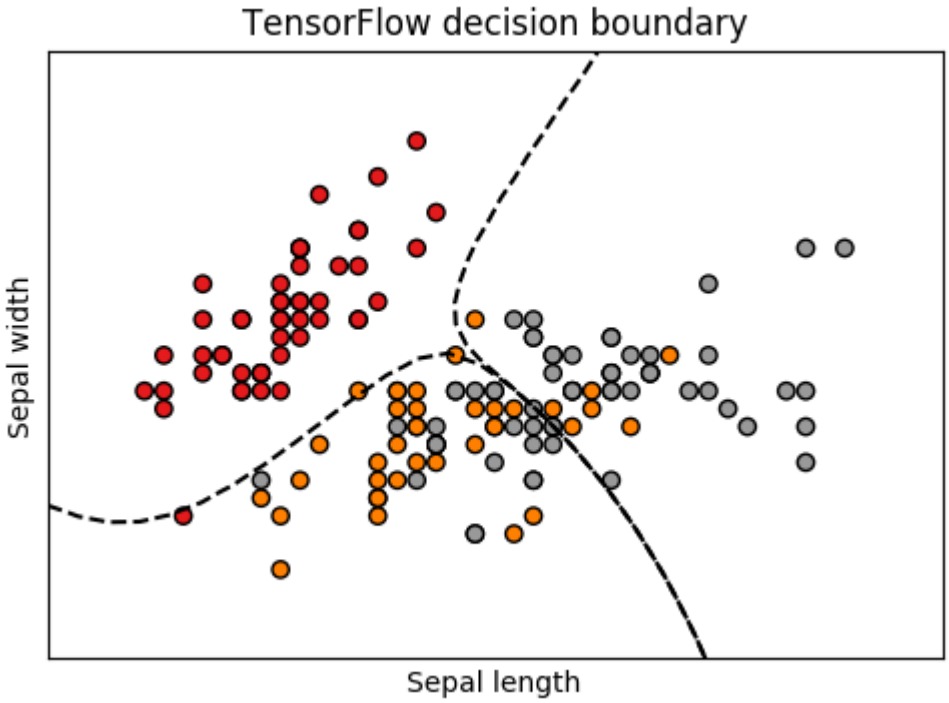
While KNN is commonly used in supervised learning for classification and regression tasks, it can also be applied in unsupervised settings such as clustering. We use the idea of Nearest Neighbors for article retrieval. Each article is represented using TF-IDF representation.

For example, if the user is currently reading article: `The dollar slipped broadly on Friday as traders booked profits after recent gains but the U.S. currency remained well-placed for further advances, supported by strong U.S. economic data that has prompted markets to dial back expectations for interest rate cuts.`, The following recommendations are generated:

```
You might also like:

Recommendation 1: Dollar Rises on the Interest Rate Plays  NEW YORK (Reuters) -
The dollar rose on Thursday as  traders, short of dollars after relentlessly
selling them for  weeks, looked to the growing yield advantage of U.S. assets as
a reason to buy back the currency before the end of the year.

Recommendation 2: Dollar's Gains Cut as Fed Raises Rates  NEW YORK (Reuters) - The
dollar's gains were clipped  on  Tuesday as the Federal Reserve raised interest
rates for the  fifth time this year, as expected, but quashed hopes for more
aggressive rate tightening.
```

We observe that the top articles recommended are close to the topic which user is currently reading. The TF-IDF representation allow us to focus on key words which are common locally but rare globally.

Naive Bayes



Introduction

Naive Bayes is a classification algorithm based on probability theory. Unlike algorithms such as perceptron or linear regression, which focus on learning decision boundaries, Naive Bayes learns classifiers based on Bayes' rule.

Assumption

Naive Bayes makes a very bold assumption about the data, it assumes that the features are conditionally independent. This assumption, which is why the model is called "naive," says that given a class label, the presence or absence of one feature is independent of the presence or absence of any other feature. Assume a scenario where symptoms like fever and chill indicate a person has malaria. Now, given a person has malaria, we know the person has a fever. Of course fever and chills are dependent on each other in some way, but the knowledge of malaria is enough to tell us whether the person has a fever or not, chills become redundant knowledge.

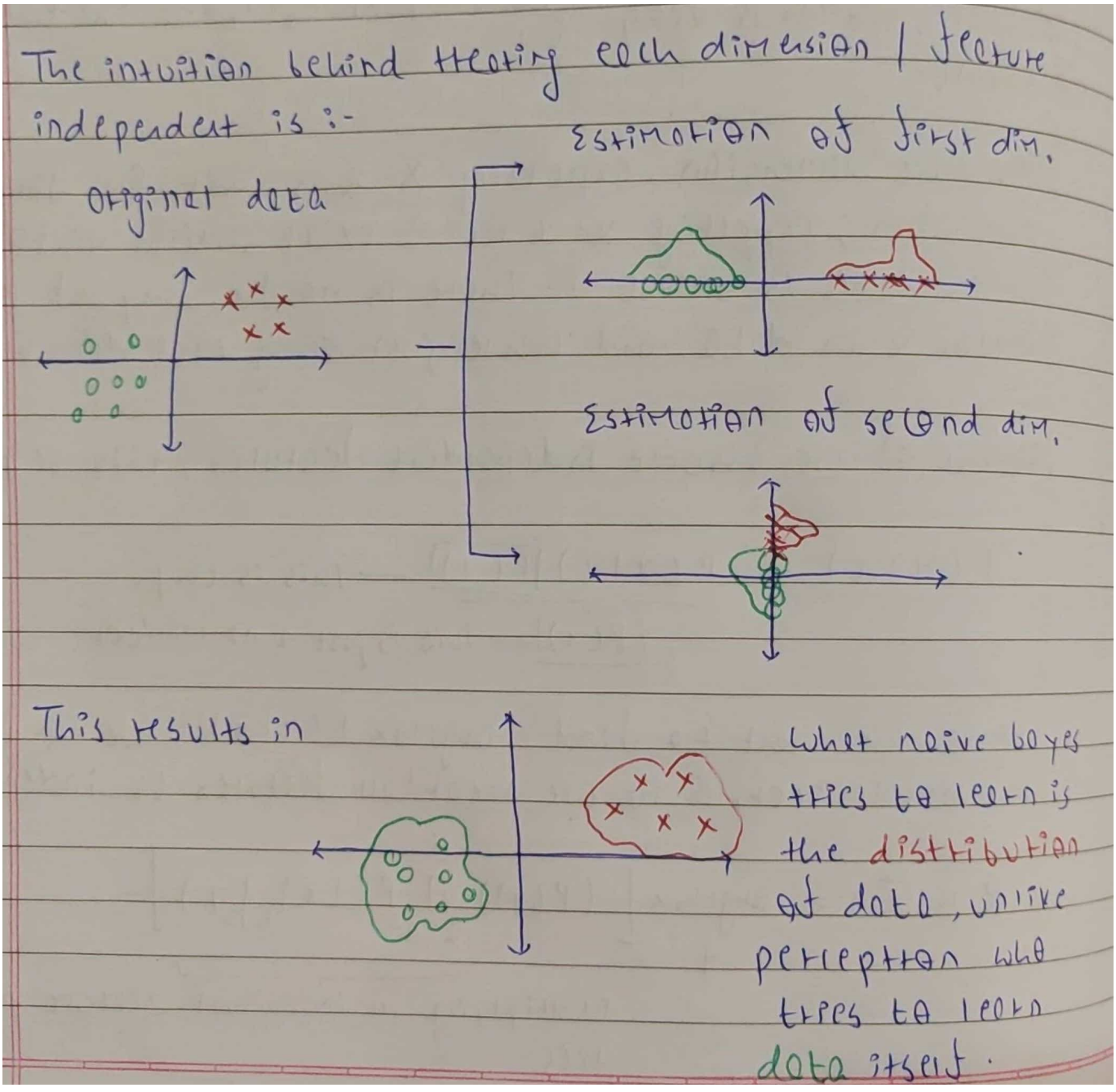
Why make such an assumption?

The whole idea of learning classifiers based on Bayes Rule is that these are generative models. The goal of generative models is to learn the underlying distribution of data. This idea is different from discriminative models like perceptron which learn how to distinguish the different classes by learning a decision boundary. In simpler terms, learning classifiers based on Bayes Rule doesn't directly model the probability of a class (Y) given the data (X) which is `(P(Y | X))`. Instead, it focuses on the probability of the data (X) given a specific class (Y) which is `(P(X | Y))`. This essentially represents the likelihood of encountering a particular data point considering its class membership. Thus, if we look at classifiers with the Bayes rule, then each `P(X = Xa | Y = c)` is a parameter.

Consider a simple feature vector `X` with n features, where each feature is a boolean value, and 2 classes `{+1, -1}`; we need to estimate approximately `2^(n+1)` parameters. Even for such a simple scenario, the parameters scale exponentially with the number of features. It just becomes impractical to estimate these parameters from data. This whole idea of conditional independence allows us to reduce parameters from `2^(n+1)` to `2n`. This is the key to Naive Bayes. This also simplifies the math to:

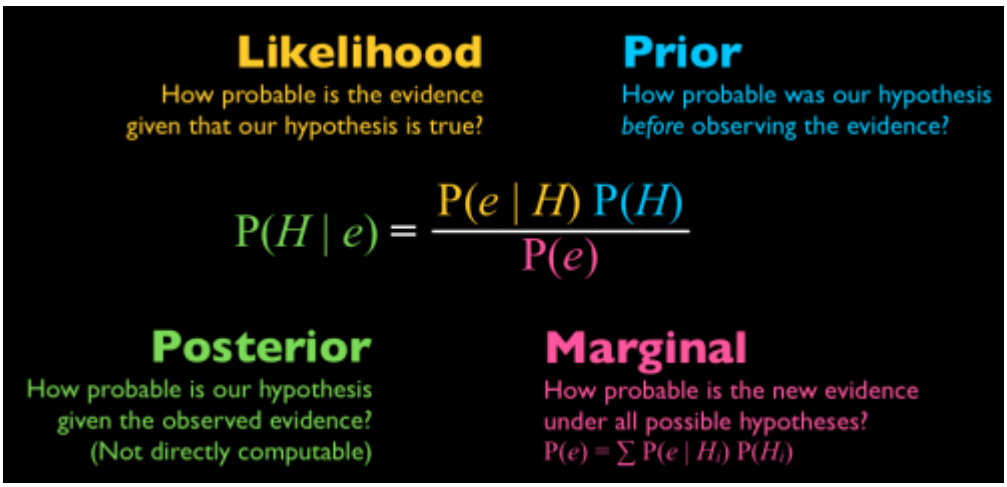
```
P(X|Y) = [P(x1|Y) * P(x2|Y) * ... * P(xn|Y)]

P(Y|X) = [P(x1|Y) * P(x2|Y) * ... * P(xn|Y)] * P(Y) / P(X)
```

Calculating P(Y) is easy - just count how many examples you have for each class and make sure they all add up to one. As for P(X), it's like a constant that doesn't really change our guess. The big change comes when we break down `P(x1|Y) * P(x2|Y) * ... * P(xn|Y)` instead of directly dealing with `P(X|Y)`. It's like shifting from searching in many dimensions to just one, which is much easier.



Algorithm

During prediction, when we need to determine `P(Y | X)`, Naive Bayes uses Bayes theorem to arrive at the desired probability.



Here, each term holds a specific meaning:

`P(H)`: This represents the prior probability of class H occurring in the dataset. It reflects the inherent bias towards a particular class if it exists. `P(e)`: This denotes the prior probability of encountering data point X itself, independent of any class. `P(e | H)`: This term, known as the likelihood, is crucial. It represents the probability of observing data point X given that it truly belongs to class Y. This value is estimated using the training data.

When working with Naive Bayes, we typically encounter calculating probabilities for three types of scenarios:

- Categorical
- Multinomial
- Continuous

Categorical

```
P (Xa = A | Y = c) = (# number of samples with class c and feature Xa as A) / (# total number of samples in class)
```

Multinomial

```
P(Xa | Y = c) = (# number of times feature X appears in instances of class c)/(# total occurrences of all features for class c)
```

Continuous

In the case of continuous features, such as numerical data, the probability density function (PDF) is often assumed to follow a specific distribution, commonly the Gaussian (normal) distribution due to its simplicity and wide applicability. The Gaussian distribution is characterized by its mean (μ) and variance (σ^2).

For each class c, the Naive Bayes classifier estimates the parameters of the Gaussian distribution for each feature. Let's denote the feature as x. Then, the probability density function (PDF) of x given class c can be expressed as:

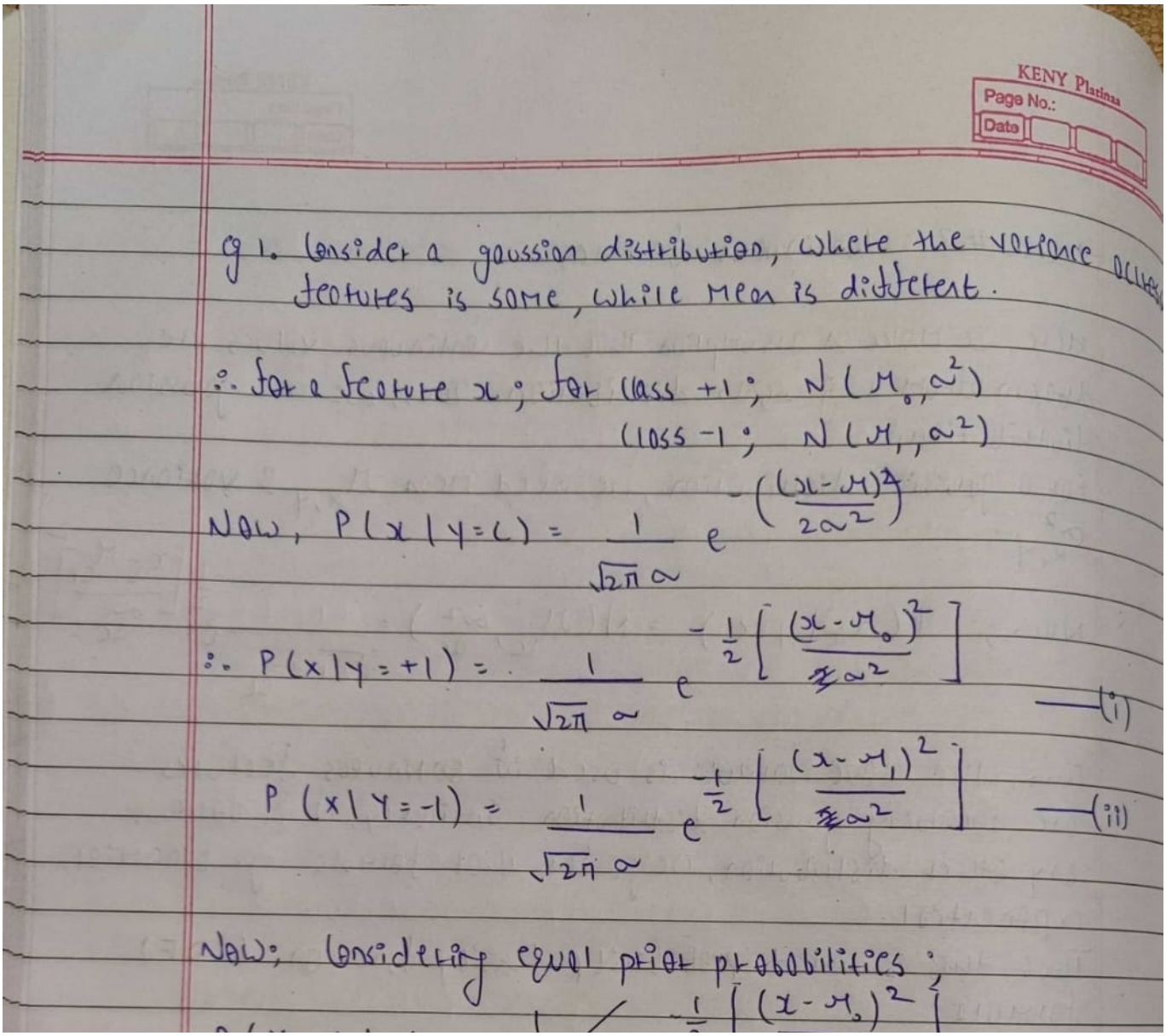
$$P(x|c) = \frac{1}{\sqrt{2\pi\sigma_c^2}} \exp\left(-\frac{(x-\mu_c)^2}{2\sigma_c^2}\right)$$

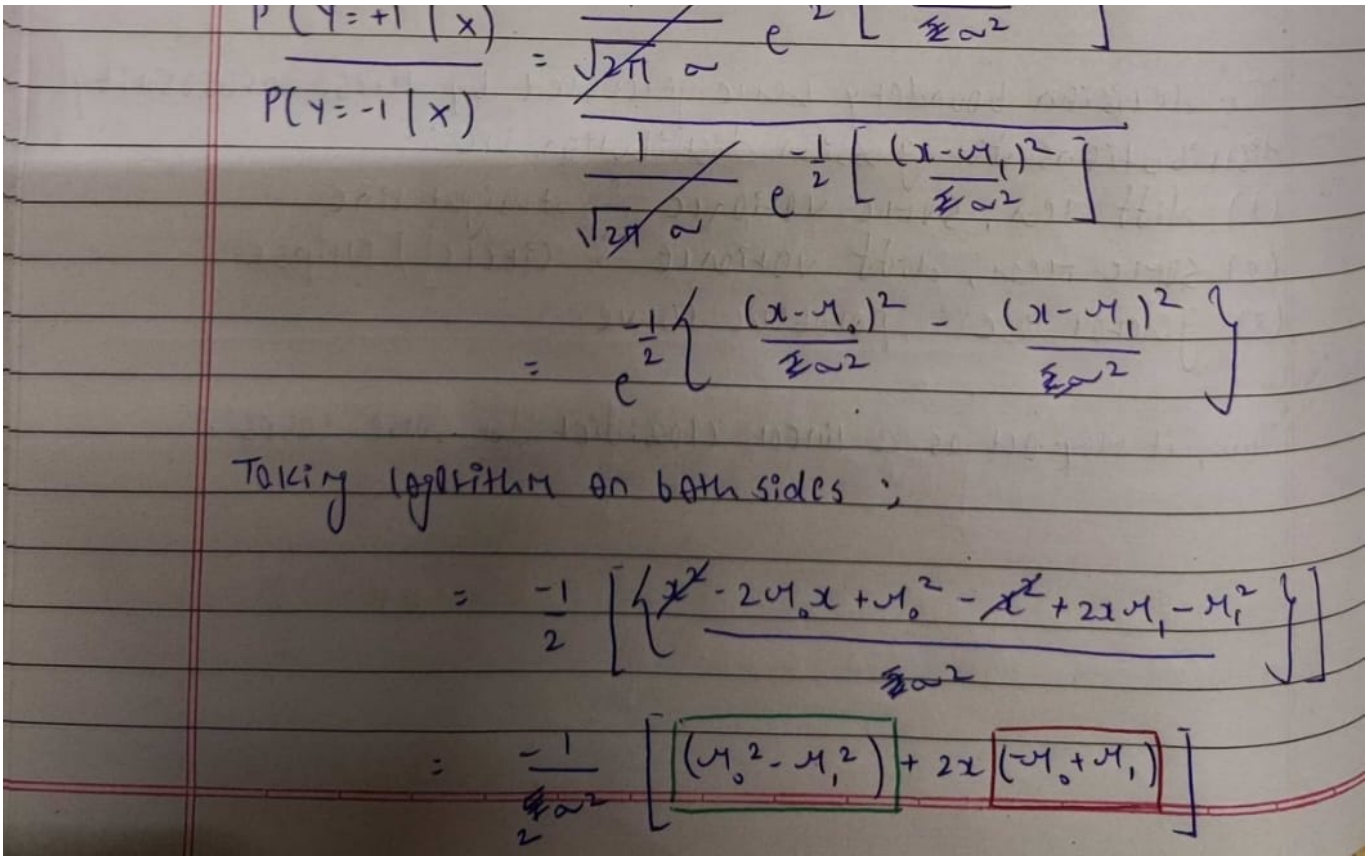
Where:

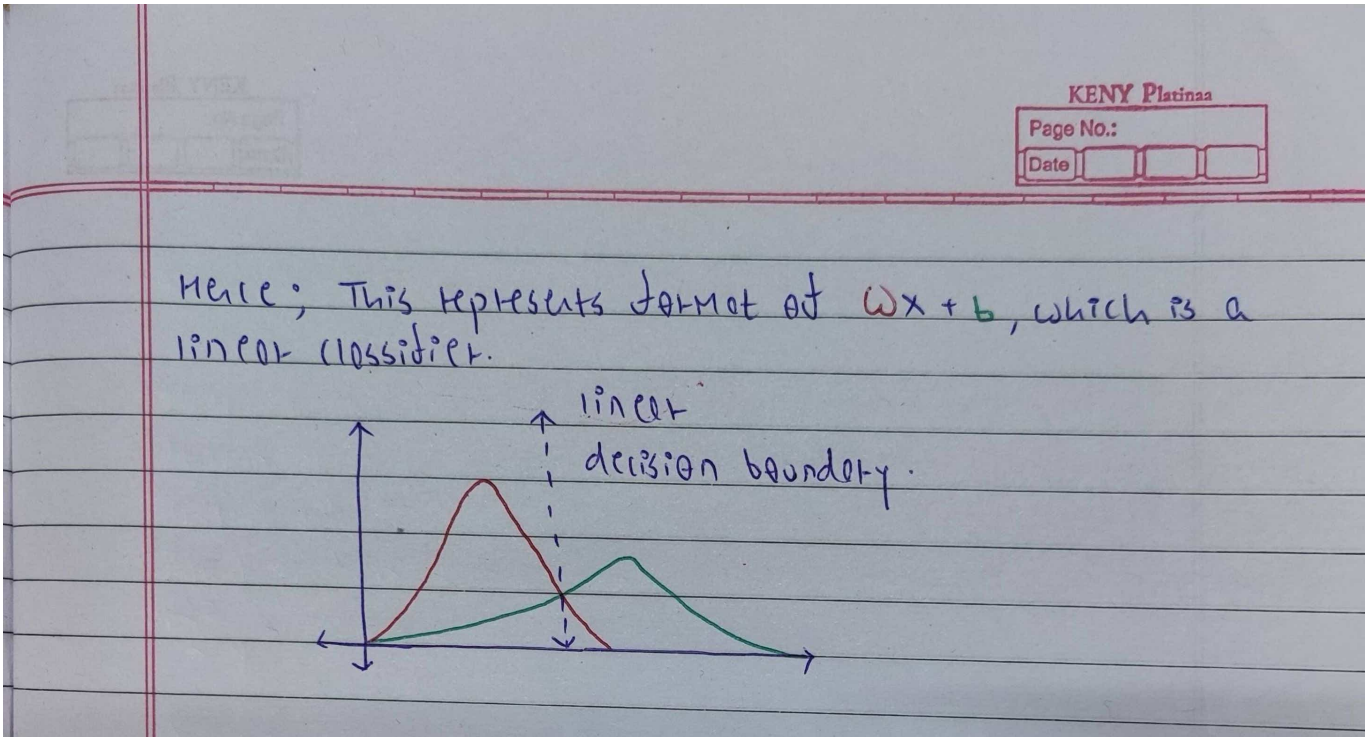
- μ_c is the mean of feature x for class c.
- σ_c^2 is the variance of feature x for class c.
- π is the mathematical constant pi.

Naive Bayes as a linear classifier

While Naive Bayes is a generative model, it can resemble a linear classifier in certain scenarios. By learning the underlying probability distribution, Naive Bayes indirectly learns a decision boundary between classes.







Relationship Between Naive Bayes Classifiers and Logistic Regression

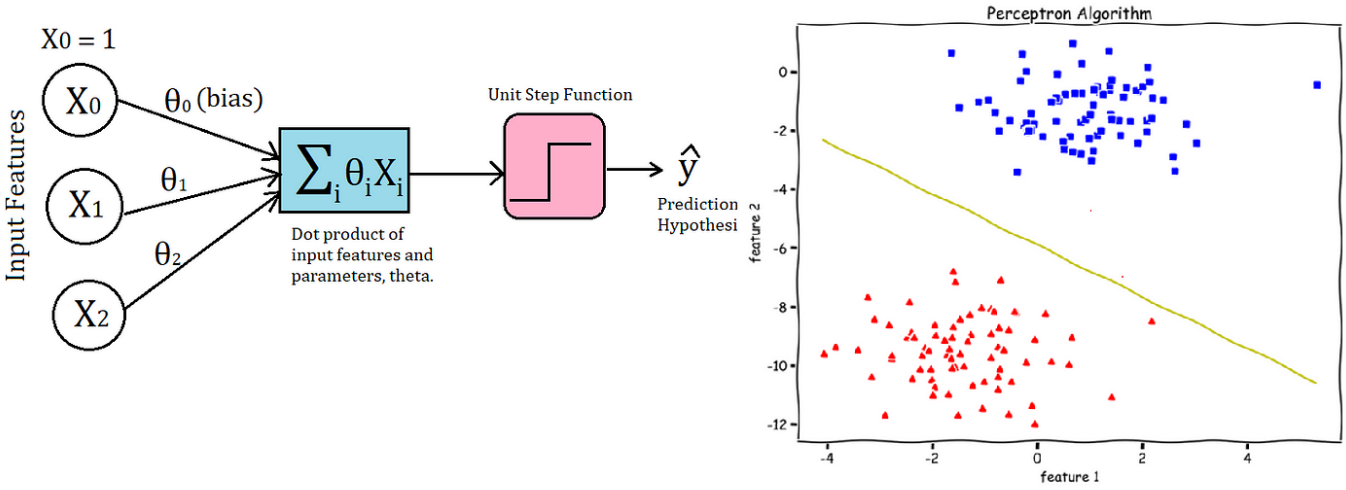
Both Naive Bayes and Logistic Regression aim to learn the relationship between features and class labels. However, they approach this task differently. Naive Bayes focuses on estimating the underlying data distribution by assuming conditional independence, while Logistic Regression directly estimates the parameters of the class distribution. Under certain conditions, such as in Gaussian Naive Bayes, the two approaches can be equivalent, learning the same decision boundary.

Proof for the same can be found here.

Results

We extract features such as last character, bigrams and trigrams from the first name to construct probabilities. Naive Bayes achives an accuracy of `88.46%` on the test data which is slightly higher than `86.92%` by logistic regression. Naive Bayes might be better for gender prediction task because the size of dataset available is small.

Perceptron



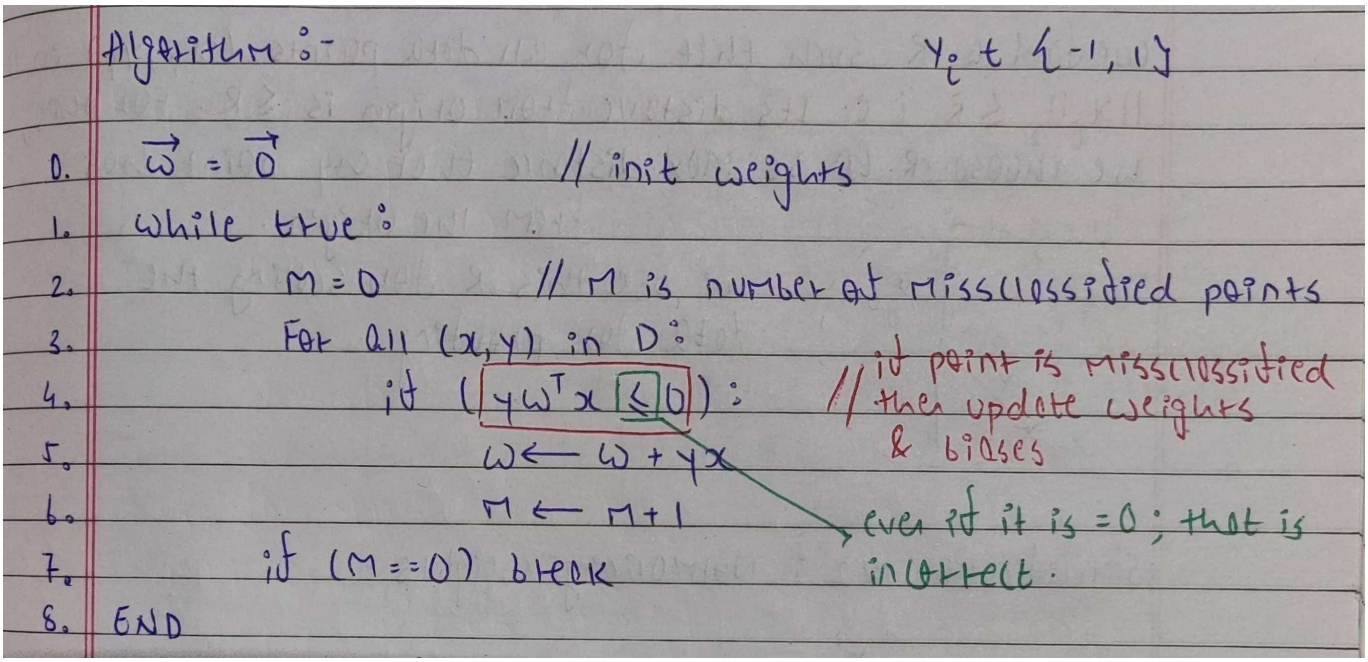
Assumptions

Perceptrons are used for binary classification. They assume that there's at least one straight line (or plane, in higher dimensions) that can perfectly separate the two classes. While this isn't always true in low dimensions, it tends to hold in higher dimensions because points are more spread out, making separation easier. So, for low dimensions, other algorithms like KNN are preferred, while for higher dimensions, perceptrons are useful.

Algorithm

In a perceptron, we define a hyperplane using a weight vector `w` (normal to the hyperplane) and a bias `b`. We can combine these into one vector `W = [w, b]` by extending our feature space into one higher dimension. Essentially, we're absorbing the bias term as one dimension and lifting our points into a higher dimension.

A hyperplane is like a flat sheet, one dimension lower than the feature space. For instance, in 2D, it's a line. Instead of characterizing it with both `w` and `b`, we extend it to 3D and draw a plane passing through the origin, removing the need for an extra bias term.



During inference, we look at the direction of the point relative to the hyperplane using `W.T @ x` and classify based on the sign of the result. If `W.T @ x > 0`, it's in the positive class, and vice versa.

However, if the data isn't linearly separable, the algorithm will keep trying indefinitely. To avoid this, we can set a limit on the number of iterations.

Proof that Perceptron will always converge

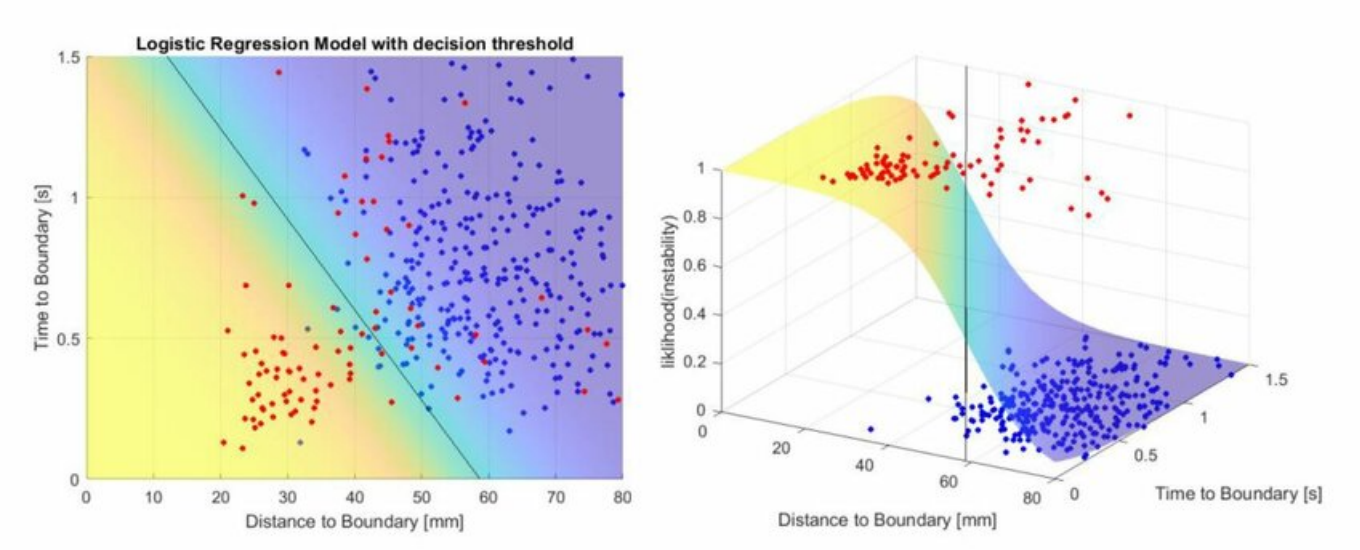
If the points are linearly separable (can be perfectly separated by a line or hyperplane), the Perceptron will converge to a solution (though not necessarily the best one). But if they're not separable, it will loop forever.

Click Here for proof that the Perceptron will always converge if the data fits our assumption.

Results

The Perceptron algorithm for Gender Classification converges in 58 steps when we consider features such as the last character, bigrams, and trigrams from the name. It achieves an accuracy of `86.15%` on the test data. However, when we simplify the features to only include the last character and bigrams, the algorithm struggles to converge. This indicates that the Perceptron excels in higher dimensions, where it can more effectively distinguish between classes.

Logistic Regression



Introduction

Logistic regression is a linear classifier and discriminative counterpart of Naive Bayes. It tries to learn `P(Y | X)` directly. This is different from Naive Bayes which learns `P(X | Y)` and `P(Y)`.

Consider the example of classification between cats and dogs. A generative model like naive bayes tries to understand what features makes an animal dog. Thus it tries to draw an image of what a dog might be. Discriminative model on the other hand tries to find the features which can distinguish a dog from cat. To put it simply, Generative model tries to understand what makes an animal a dog or a cat, while Discriminative model focuses on pinpointing the key features that tell them apart.

Assumption

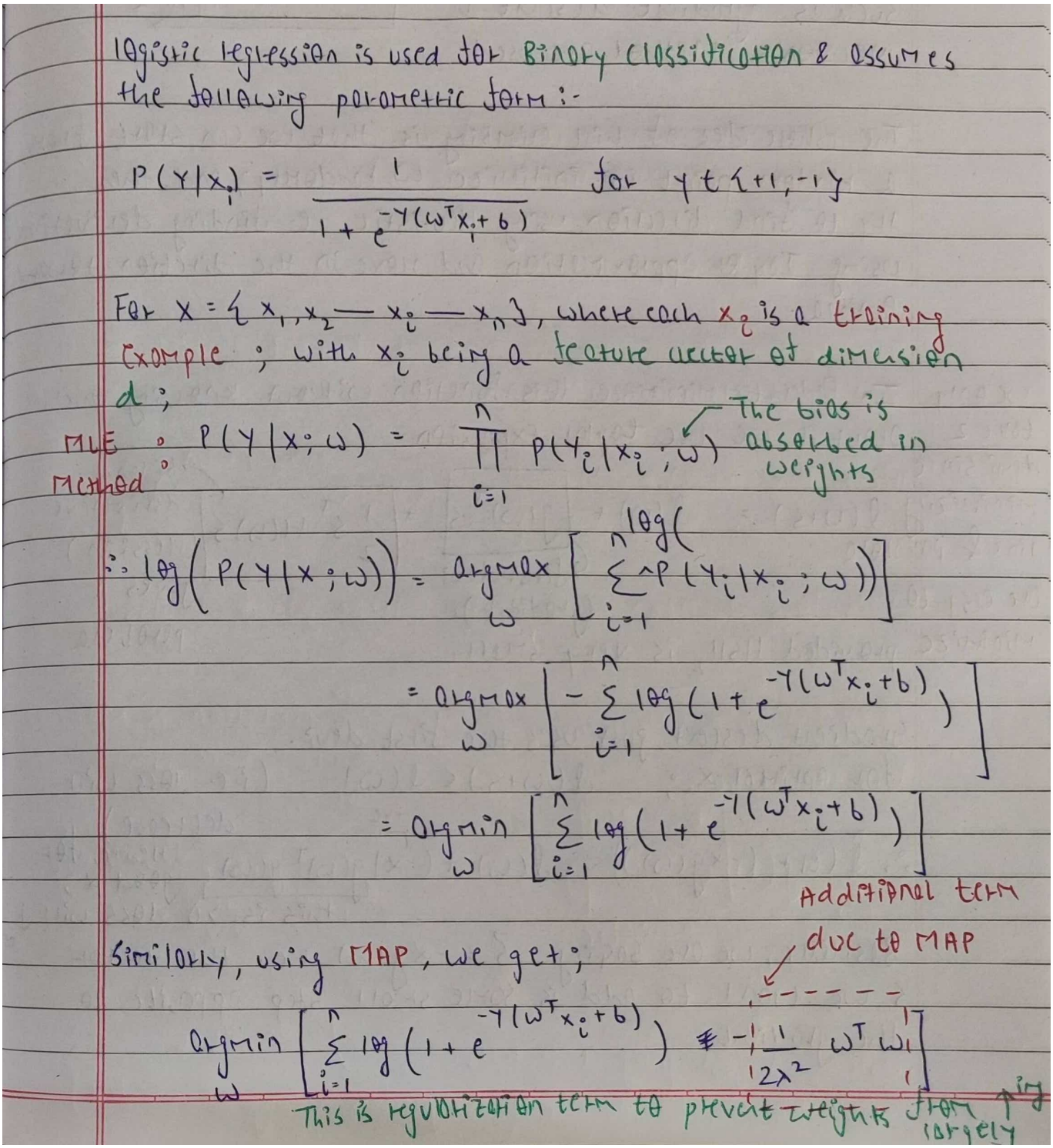
Logistic regression is comparatively more flexible and does not make any strict assumptions about the data. It generally works well for data drawn from exponential family distribution. The exponential family consists of several common probability distributions, including the normal (Gaussian), exponential, Poisson, binomial, and gamma distributions, among others.

Algorithm

Every discriminative algorithm assumes a certain parametric form over `P(Y|X)` and tries to learn the parameters for this directly by maximum likelihood (MLE) or Maximum A Posteriori (MAP) estimation. This parameteric form represents the sigmoid function.

$$P(y|\mathbf{x}_i) = \frac{1}{1 + e^{-y(\mathbf{w}^T\mathbf{x}_i + b)}}.$$

MLE tries to maximize the probability of distribution given the data, that is it tries to maximize `P(D|θ)`. MAP on other hand places a prior probability over parameters `P(θ)` and tries to maximize the probability of parameters given the data, that is `P(θ|D)` which is nothing but `P(D|θ)P(θ)`. This involvement of extra term `P(θ)` helps prevent overfitting by introducting regularization term.



For both MLE and MAP, we arrive at a step which does not have a closed form solution. However, one of the nice properties of this function is that it is convex, continuous, and differentiable in nature. This implies we can use algorithms based on Hill Climb Search such as Gradient Descent or Newtons Method to find the argmin.

The whole idea of Hill Climbing is that we start randomly, that is we initialize `W` and `b` randomly such as with `0`s and then we find the direction towards the optima using some heuristics. Here, we use taylor approximations to provide us with that heuristic. Once we have found the direction, we take small steps (for taylor approximation to hold) in that direction.

Click Here to learn about Logistic Regression using Gradient Descent and Newton's Method.

Naive Bayes vs Logistic Regression

One key difference of Logistic Regression and Naive Bayes is that Logistic Regression is much for flexible. It does not makes overly bold assumptions about the conditional independence of features like Naive Bayes. We do assume a parametric form for logistic regression, but do make any assumptions about data, which allows the probability distribution to be any one of the exponential family. This allows logistic regression to be more flexible, but such flexibility also requires more data to avoid overfitting. Typically, in scenarios with little data and if the modeling assumption is appropriate, Naive Bayes tends to outperform Logistic Regression. However, as data sets become large logistic regression often outperforms Naive Bayes, which suffers from the fact that the assumptions made on `P(x|y)` are probably not exactly correct.

Imagine two features that are highly correlated, like having the same feature counted twice. Naive Bayes treats each copy of the feature as separate, which can lead to overestimating the evidence. However, logistic regression handles correlated features better. If two features are perfectly correlated, logistic regression splits the weight between them, leading to more accurate probabilities when there are many correlated features.

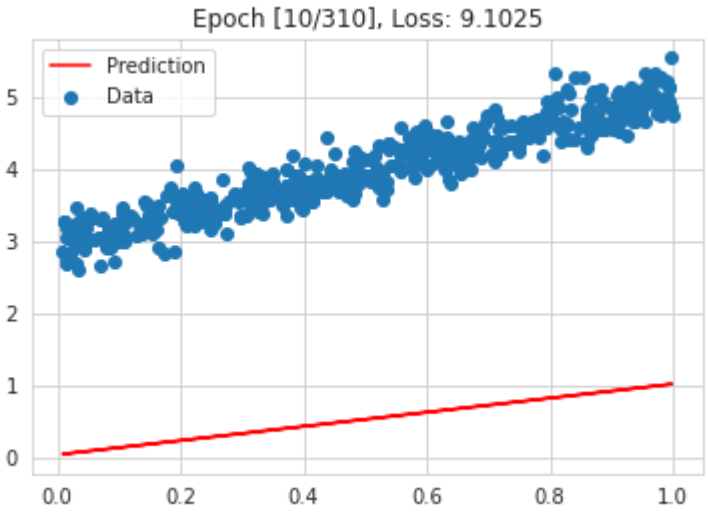
Given all the assumptions hold, Naive Bayes converges faster than Logistic Regression.

Results

The logistic regression model trained on the gender dataset achieves an accuracy of approximately `90%` on the test set. Guessing gender from names is little tricky, even for humans, because names vary a lot and can be confusing. So the model does a decent job in predicting genders from names alone. We can also consider features such as trigrams, quadgrams from names to potentially improve the accuracy.

The accuracy obtained is slightly higher than `88.46%` by Naive Bayes. This might be because of the flexibility of Logistic Regression over Naive Bayes in terms of the assumptions made over data distribution.

Linear Regression



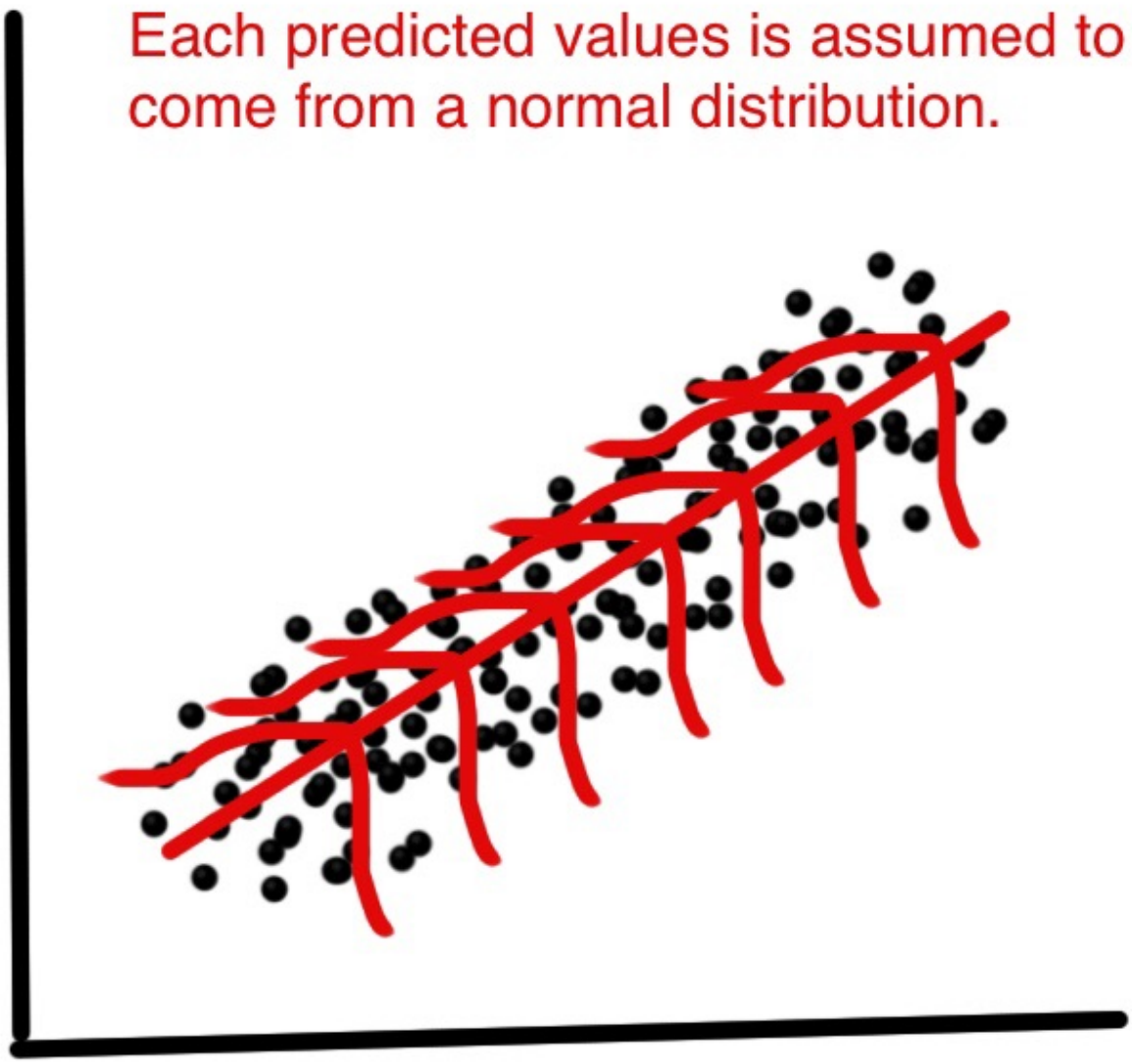
Introduction

Linear regression allows us to predict a continuous value (like house price) based on one or more factors (like house size) by fitting a straight line through the data. Unlike classification tasks where labels fall into distinct categories, linear regression handles problems where the label is a real number.

Assumptions

Linear regression makes two core assumptions about the data. First, it assumes a linear relationship between the features and the label. For example, house price might have a linear relationship with the area of the house. Second, it assumes the errors/noise (the difference between actual values and the fitted line) follow a Gaussian distribution. This means the data points will be scattered around the true linear regression line in a bell-shaped curve.

Thus, we can formalize these assumptions as $y = WX + b + \varepsilon$ where ε is the Gaussian distributed error or noise, that is $\varepsilon \sim N(0, \sigma 2)$ where $\sigma 2$ is the variance of the noise.



In simpler terms, this equation also implies that `y` itself follows a normal distribution with a mean of `WX + b` and a variance of `σ^2`. We can represent this as: `y ~ N(WX + b, σ^2)`

Algorithm

Given `y ~ N(WX + b, σ^2)`, the goal of linear regression is to estimate the parameters `W` and `b`. This can be done either by maximum likelihood (MLE) or Maximum A Posteriori (MAP) estimation.

MLE vs MAP

Imagine you find popcorn on your living room floor. There are three possible scenarios: watching a movie, playing board games, or sleeping. MLE aims to find the scenario that is most likely to result in popcorn on the floor. So, it maximizes the probability of popcorn given each scenario and selects the one with the highest likelihood. Watching a movie in living room is more likely to lead to popcorn on the floor, and hence MLE would pick that event.

MLE works by maximizing the likelihood of the model given the data, that is it tries to maximize $P(D|\theta)$. It's like fitting a line to the data points in such a way that it maximizes the chance of observing those data points. This idea gives us the Ordinary Least Squares (OLS) loss function. OLS minimizes the squared differences between the predicted and actual values, increasing the likelihood of the model fitting the data.

However, MLE might not always give the most sensible answer. For instance, if we introduce another event like a popcorn throwing competition, MLE might wrongly suggest that this event is most likely to cause popcorn on the floor. This is where MAP comes in. MAP considers prior probabilities of events. It factors in our prior knowledge or beliefs about the likelihood of each event. So, even if a popcorn throwing competition could theoretically lead to popcorn on the floor, our prior belief that watching a movie in living room is much more common helps us choose the more likely event.

MAP places a prior probability over parameters $P(\theta)$ and tries to maximize the probability of parameters given the data, that is $P(\theta|D)$ which is nothing but $P(D|\theta)P(\theta)$. It combines the likelihood of the data given the parameters (as in MLE) with the prior probability of the parameters. This involvement of the extra term $P(\theta)$ helps prevent overfitting by introducing a regularization term.

Click Here to view the derivation of estimation of W and b. This shows how we obtain the Ordinary Least Squares (OLS) loss function and regularization term.

$$MSE = \frac{1}{n}\Sigma\underbrace{\left(y - \hat{y}\right)^2}_{\text{The square of the difference between actual and predicted}}$$

Unlike logistic regression which uses a complex loss function, linear regression utilizes the Ordinary Least Squares (OLS) loss function. Because the OLS loss function is a parabola, it has a closed-form solution. This means we can theoretically find the optimal W and b in one step using methods like Newton's method. However, for problems with high dimensionality (many independent variables), calculating the Hessian matrix required by Newton's method becomes computationally expensive. Therefore, iterative optimization algorithms like gradient descent are preferred in practice.

Results

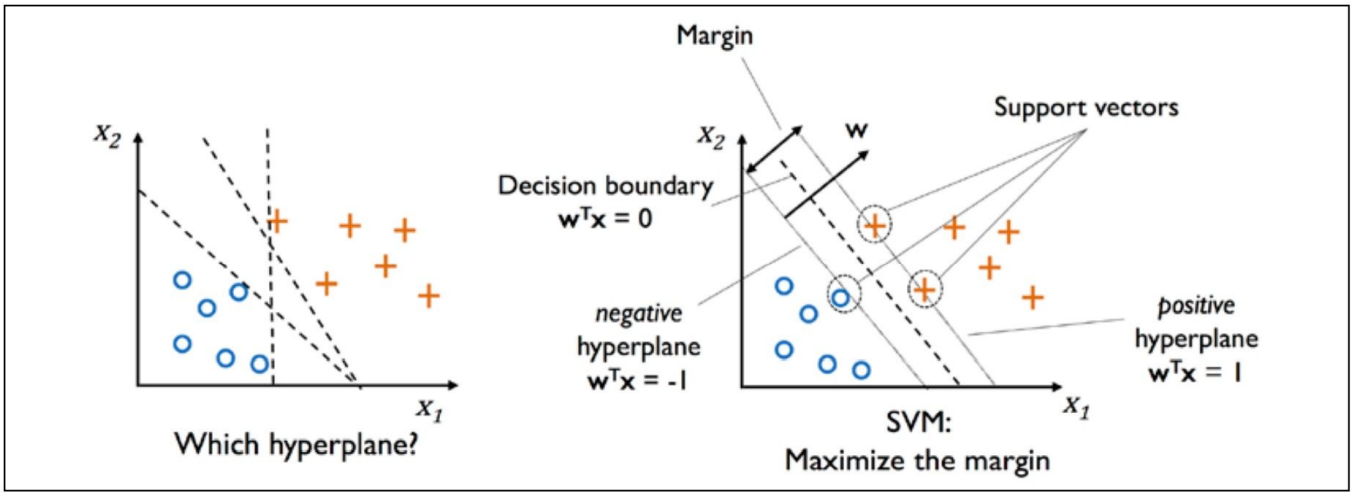
We trained a linear regression model to predict house prices in Mumbai. On the test set, our model achieved an mean squared error (MSE) of 0.7 and mean absolute error of 0.55 Crores. In simpler terms, this means the predicted prices typically differed from the actual prices by around 0.55 Crores.

Compared to other algorithms like KNN where the average absolute error was around 0.31 Crores, our linear regression model performs less accurately. This could be because house prices in Mumbai can vary greatly

depending on location. A similar-sized house in Juhu could cost significantly more than one in Borivali. Linear regression assumes a straight-line relationship between features and price, which might not capture the impact of location as effectively. KNN, on the other hand, can better handle these kinds of variations by considering similar houses that have already sold.

Additionally, linear regression uses squared loss function, which represents mean. Median is a better quantity to predict for tasks such as house price prediction, considering the impact of outliers. This can be observed by noticing that MAE and MSE are close to each other. The nature of linear regression is such that it tries to reduce the mean square error, which might not be the best thing in house price prediction since outliers are common in house prices.

Support Vector Machines



Introduction

SVM is a classification algorithm based on maximum margin linear discriminants, that is, the goal is to find the optimal hyperplane that maximizes the gap or margin between the classes.

The goal of the perceptron was to find a hyperplane, but it has no guarantee of optimality. SVM finds the optimal hyperplane by using the geometric intuition that maximizing the margin between classes would lead to better generalization.

Assumptions

SVMs assume that the data is linearly separable. However, unlike perceptron which requires this assumption to be followed strictly, SVMs can handle some noise or error in classification by introducing slack variables. This is called Soft Margin Classification.

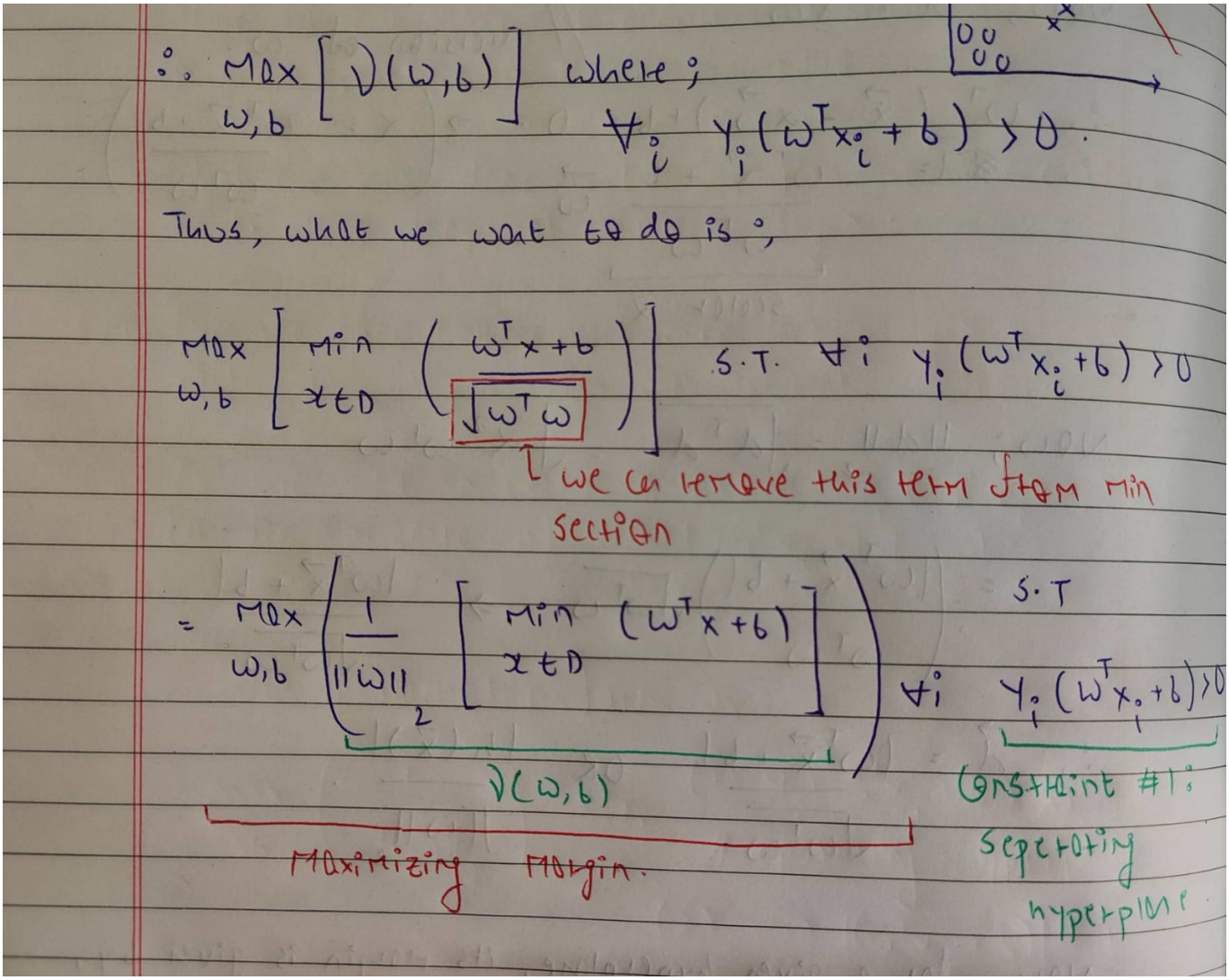
Algorithm

The distance of a point X from a hyperplane is given by

$$\delta_i = \frac{y_i \, h(\mathbf{x}_i)}{\|\mathbf{w}\|} = \frac{y_i(\mathbf{w}^T\mathbf{x}_i + b)}{\|\mathbf{w}\|}$$

It is important to note that the numerator represents the absolute distance while the denominator performs normalization to provide relative distance.

Our goal is to find `W` and `b` that maximize margin, where the margin is the distance of the closest points to the hyperplane. This can be represented as a constrained optimization problem.



Now, we know that the numerator `| W^T.X + b |` represents the absolute distance of point X from the hyperplane. This becomes relative distance when it is normalized by the norm of `W`.

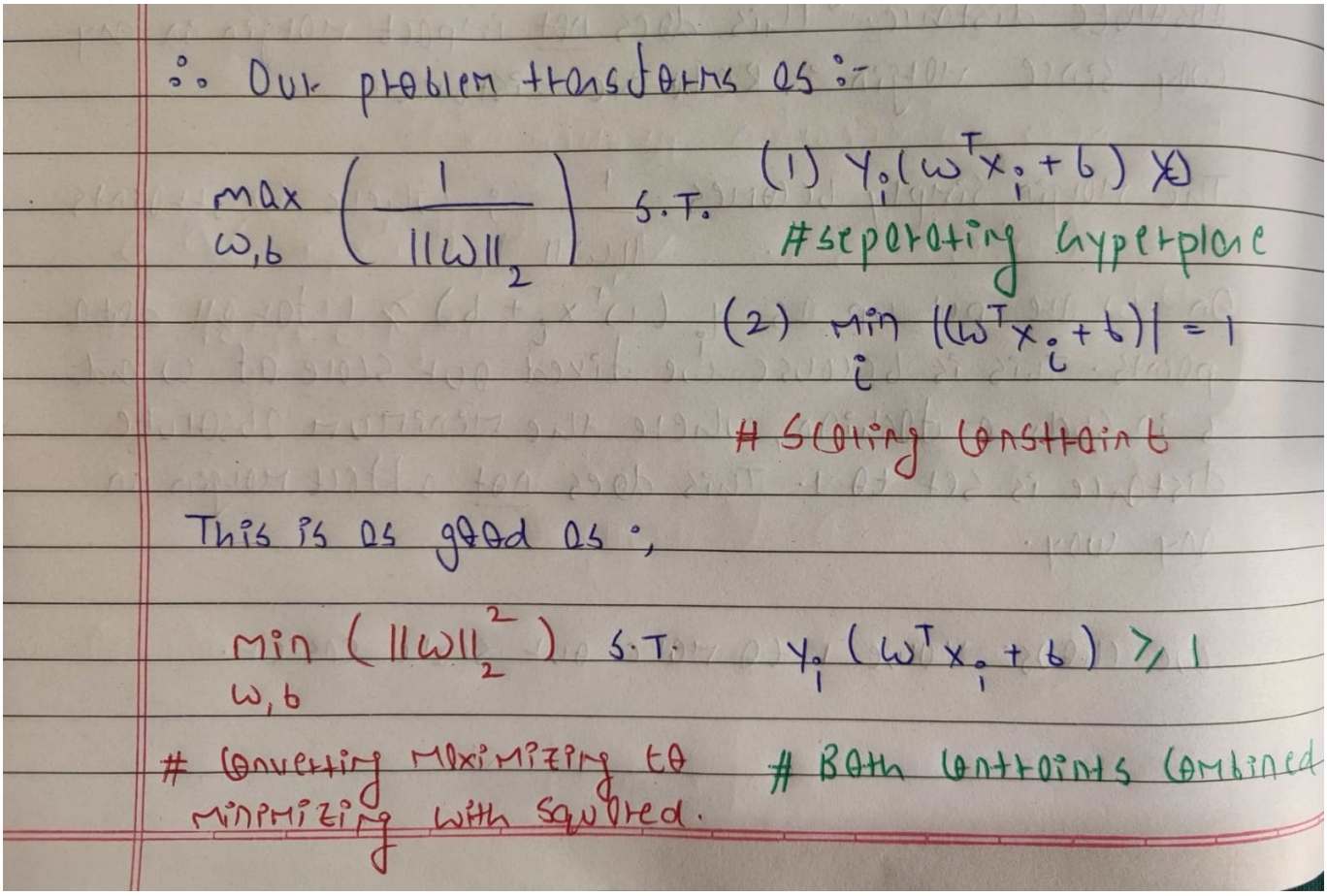
When we talk about margin, we refer to the relative distance, that is, how well the hyperplane separates two classes.

In the following proof, we can use this idea to set the `min | W^T.X + b |` to `1`. What we are trying to do is to fix a scale for the absolute distance. This does not impact the margin in any way since the margin is defined relatively.

The margin simply becomes `1/||W||`, where the new constraint on the hyperplane is that all points at least have an absolute distance of `1` from the hyperplane. This constraint has to be introduced because of scaling. We chose the value 1 since it makes our problem easier.

Consider the analogy of a pepper-eating contest between two people, A and B. A can eat things that are twice as spicy as what B can eat. Thus, for a given spice level, if B can eat 5 peppers, A can eat 10 peppers. Tomorrow, if the competition introduces spicier peppers, let's say B can now only eat 2.5 peppers. At the same time, A's limit also scales down proportionately to 5 peppers. The absolute difference between A and B has changed, but the margin remains the same: A can eat twice as spicy as B. What we have done in SVMs is fix

the spice level at 1 and introduce it as a constraint. This does not change the relative difference or margin between classes.

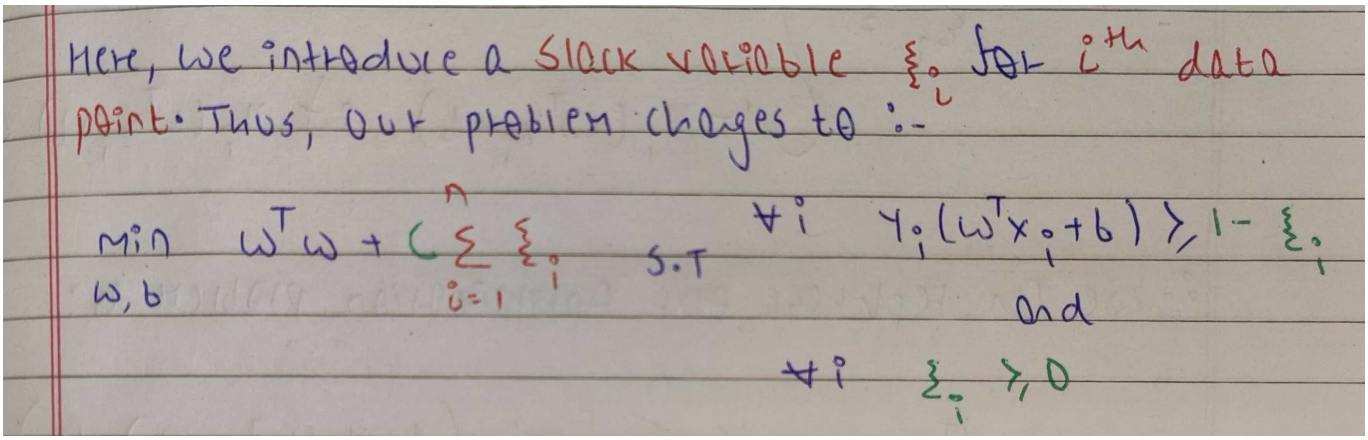


This new formulation is a Quadratic Optimization Problem (QP) which can be solved to obtain the optimal hyperplane. Here, Objective is the quadratic form `||w||^2` while Constraints are linear. Because the quadratic represents a parabola, the above formulation will always give a unique optimal solution, provided a hyperplane exists.

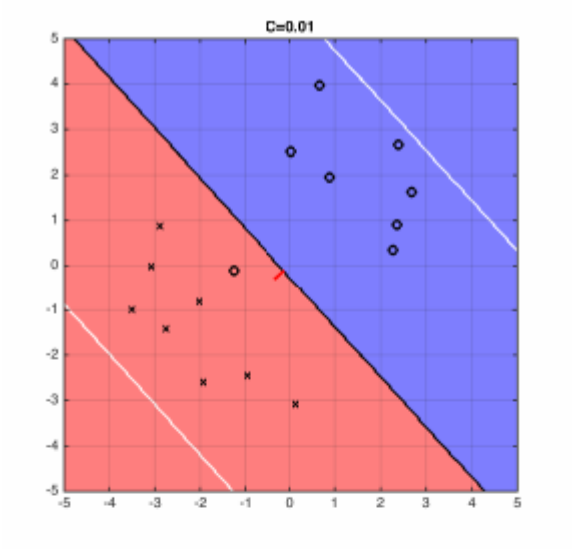
All the training points which are on the margin, are called Support Vectors. Only these points are needed to determine the optimal hyperplane. All other points can be practically ignored. Moving these support vectors in any direction will change the hyperplane but moving any other points will have no effect. This is the key property behind SVMs.

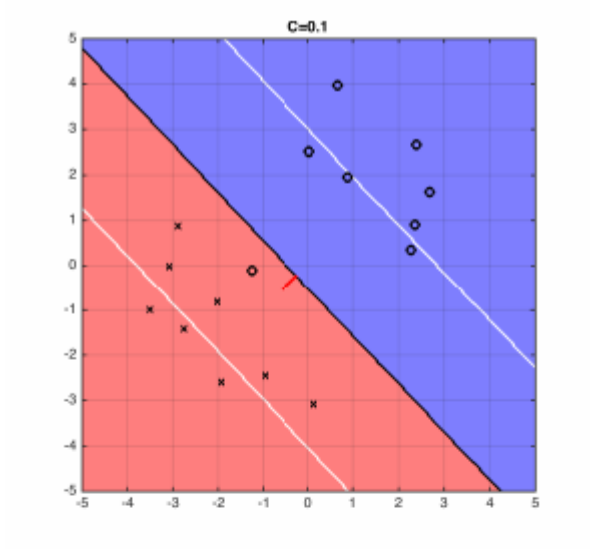
SVM with Soft Constraints

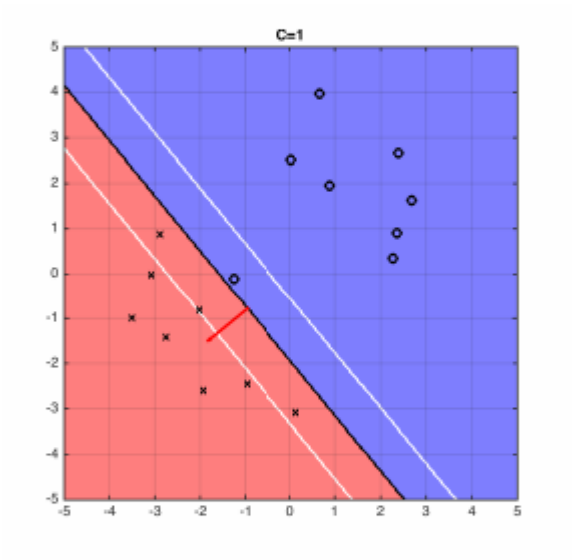
The above formulation works well when data is linearly separable. However, in real-world scenarios, data is rarely perfectly separated by a straight line. Noise or outliers can make it messy. To handle this, we introduce the concept of "slack variables" `ξi`. These slack variables allow some data points to deviate from the ideal separation line, either by being on the wrong side of the margin or by being within the margin. We do this by penalizing these deviations proportionally, and this penalty is controlled by a parameter called C.

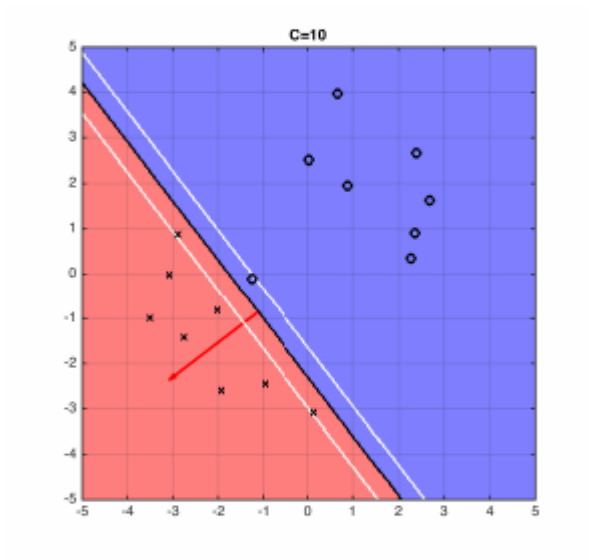


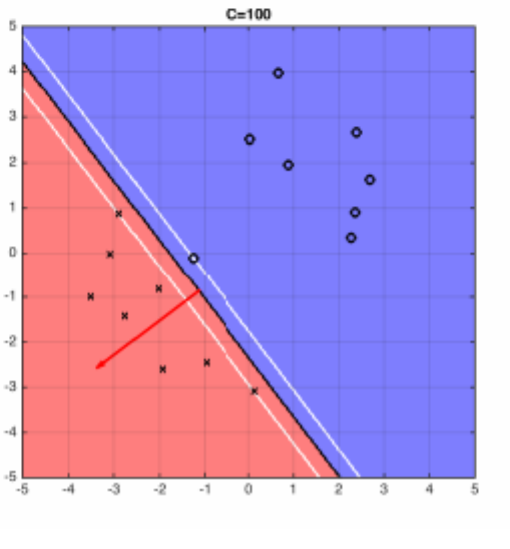
When C is large, the penalty for deviations is high. So, the model tries hard to find a separation line that includes almost all points, even if this means a very narrow margin. Conversely, when C is small, the penalty for deviations is low. This results in a wider margin, allowing more points to be on the wrong side or very close to the separation line.



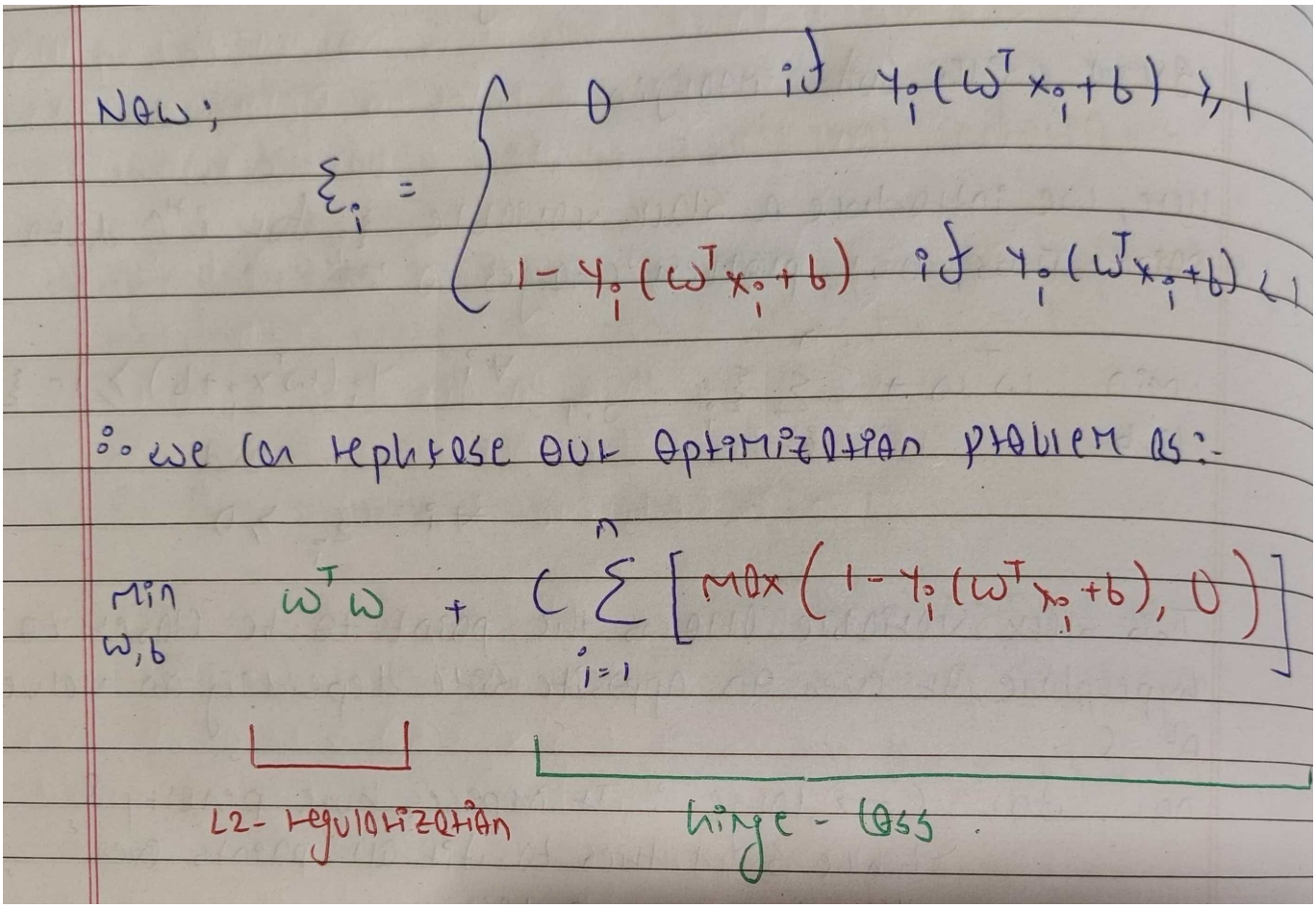








The formulation can be rephrased as:



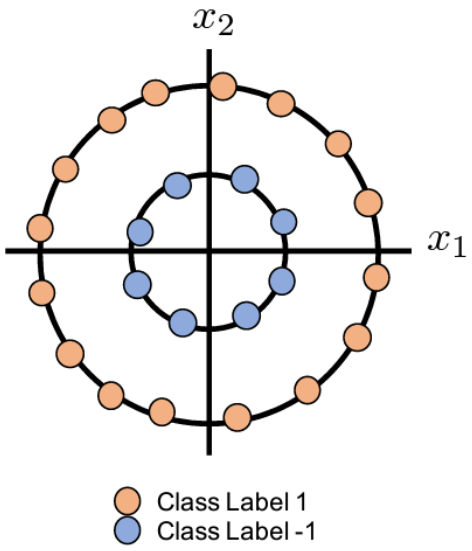
This is very similar to logistic regression, except we use hinge loss instead of logistic loss. A lot of machine learning algorithms end up with this idea of minimizing a function which is a combination of loss function and regularization term. The regularization term keeps the model simple by penalizing the complexity of the model, like the norm of weights. The loss function measures errors in data, like hinge loss here The goal is to minimize this objective function to find the optimal hyperplane Regularization term is a function of parameters, and is independent of data points, while the loss term is a function of both parameters and data. This allows us to keep the model simple as well as fit the data well.

Results

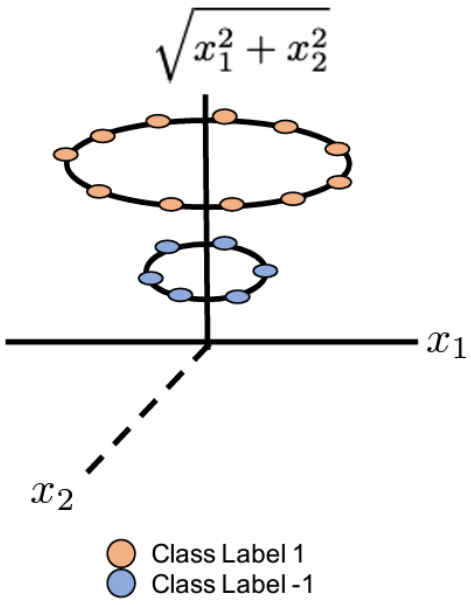
SVMs for Gender Prediction using first names provide an accuracy of `88.46%` on the test set. This is higher than the `86.15%` obtained using Perceptron. The reason behind this can be the fact that SVM chooses the optimal hyperplane. Unlike Perceptron where we need to consider the last character, bigrams, and trigrams to converge, we just required the last character and bigrams for SVMs.

Additionally, we can observe how accuracy increases over the training set as we increase our value of C from `0.01` to `1`. A larger value of C means that we are increasingly penalizing the incorrect classifications. This would increase the accuracy at the cost of margin. A small value of C would not penalize so heavily and would hence allow more error on the training set, but it will provide a bigger margin. Hence we tried to choose an optimum value of C using a validation set and found it to be around `0.1`.

Kernels



(a)



(b)

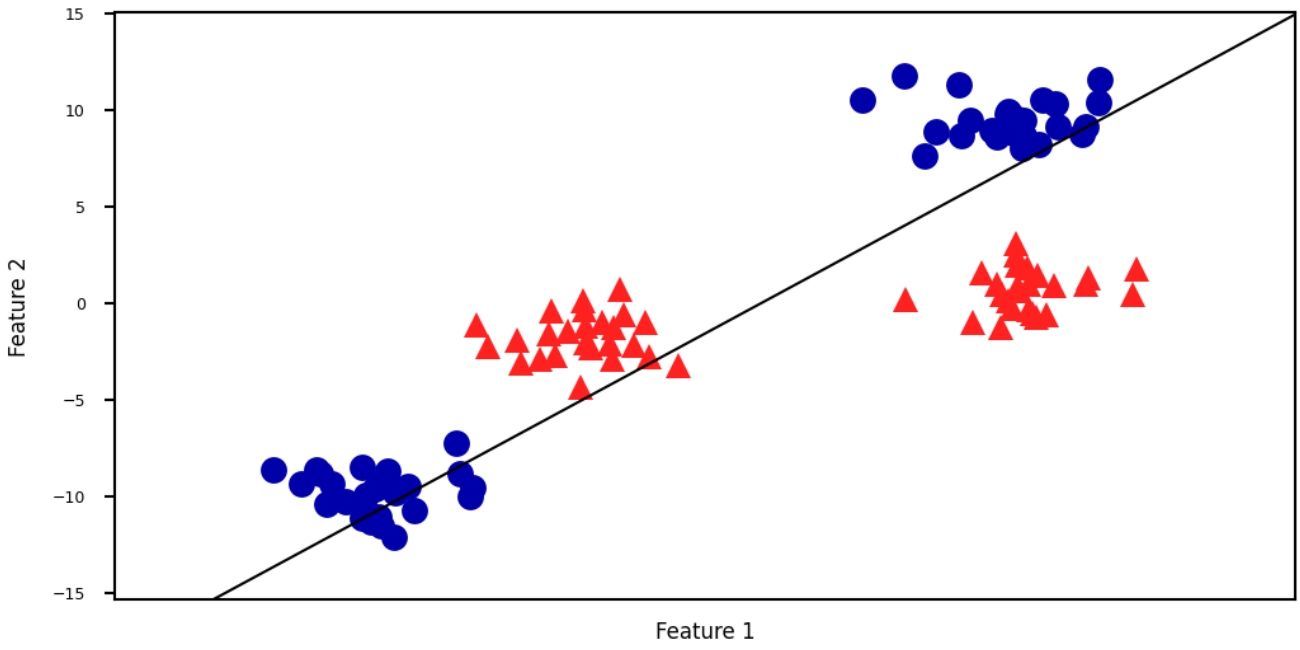
Introduction

Machine Learning algorithms like SVMs are very successful at finding linear boundaries to separate data points belonging to different classes. However, real-world data frequently exhibits non-linear relationships, posing a challenge for linear classifiers.

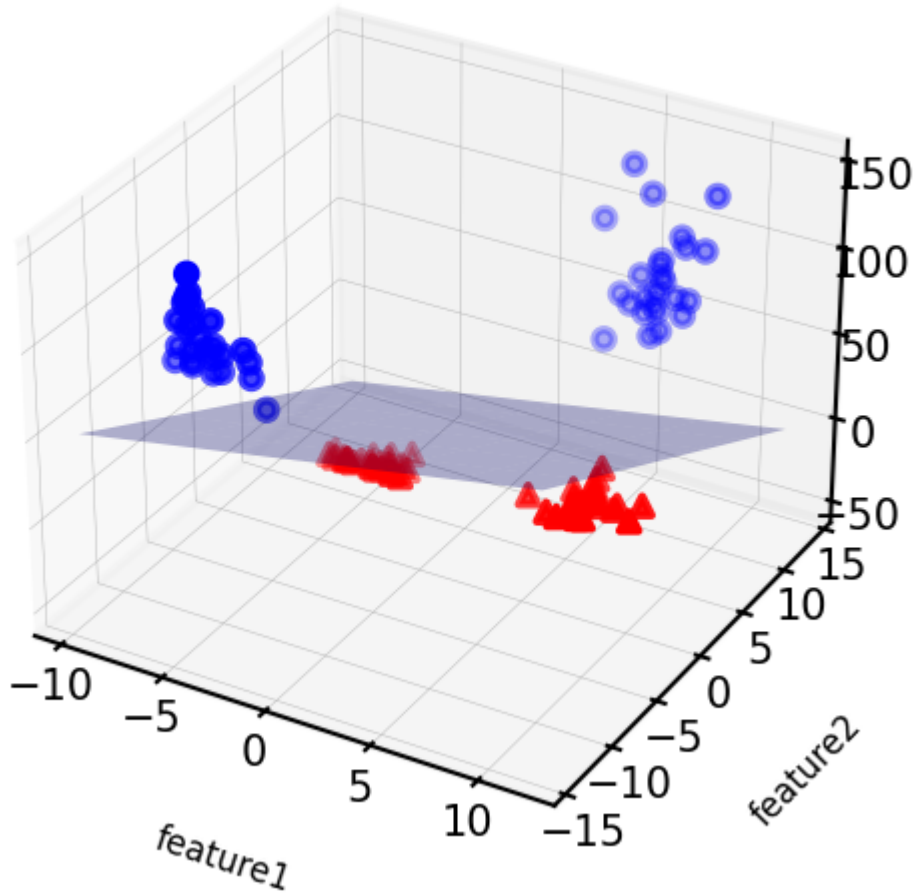
Kernelization is a technique that allows linear classifiers to learn non-linear decision boundaries. We often use kernelization when our model suffers from a high bias problem.

Intuition

Imagine a dataset with two features, say, weight and height, used to classify individuals based on body type (athletic vs. non-athletic). A linear classifier, like an SVM, would attempt to draw a straight line to separate the data points. But it is possible that some short people are athletic, and some tall people are not. A straight line simply wouldn't capture this complexity.



Kernels essentially transform the original data from a lower-dimensional space (e.g., 2D for weight and height) into a higher-dimensional space where the data becomes linearly separable. This higher dimensional spcae tries to capture the non linear interactions among the features. In our example, we might create a new feature like "body mass index (BMI)" derived from weight and height. This new, 3D space might allow a straight line to effectively separate athletic from non-athletic individuals.



The problem with simply mapping to a higher dimension is that it can get computationally very expensive. The higher dimension might consist of so many features that it is no longer feasible to store these vectors or perform computations on them. This is where kernelization makes use of the kernel trick. It's a clever way to work with the higher-dimensional space without explicitly doing any of the calculations in that higher space.

This function operates on the original data points, and calculates a similarity measure in the higher-dimensional space, without explicitly performing the mapping itself. We fit a linear classifier in higher dimension, but we never perform a single calculation in this higher dimension. The calculations are only performed on the original space.

Mathematics behind Kernel Functions

Consider the following example of mapping feature vector to higher dimension.

$$\text{Consider the following example: } \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix}, \text{ and define } \phi(\mathbf{x}) = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_d \\ x_1 x_2 \\ \vdots \\ x_{d-1} x_d \\ \vdots \\ x_1 x_2 \cdots x_d \end{pmatrix}.$$

Part I: Linear Classifiers & Inner Products

In a linear classifier, we aim to find a decision boundary that separates different classes in the feature space. This decision boundary is represented by a hyperplane. raditionally, in linear classifiers like Support Vector Machines (SVMs) or logistic regression, we represent the decision boundary using a weights `w` and bias `b`.

The decision function for a linear classifier is expressed as $f(x) = wT.x + b$. We can express the decision function solely in terms of inner products between feature vectors. The proof for the same can be found here.

What this shows us that the only information we ever need in order to learn a hyper-plane classifier is inner-products between all pairs of data vectors. Thus, if we are able to precompute the inner products and store them in a matrix `K`, all we need to do during training and testing time is to refer this matrix `K`. This idea is similar to the idea of memoization in Dynamic Programming. This matrix `K` is called as Kernel Matrix `K`. By this, we eliminate the need for `w` matrix, and instead require matrix of alphas.

Part II: Kernel Trick

For some of these feature mappings, we can find the inner product between these two vectors `φ(x)` and `φ(z)` very cheaply.

Let's go back to the previous example, $\phi(\mathbf{x}) = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_d \\ x_1 x_2 \\ \vdots \\ x_{d-1} x_d \\ \vdots \\ x_1 x_2 \cdots x_d \end{pmatrix}$.

The inner product $\phi(\mathbf{x})^\top \phi(\mathbf{z})$ can be formulated as:

$$\phi(\mathbf{x})^\top \phi(\mathbf{z}) = 1 \cdot 1 + x_1 z_1 + x_2 z_2 + \cdots + x_1 x_2 z_1 z_2 + \cdots + x_1 \cdots x_d z_1 \cdots z_d = \prod_{k=1}^{d} (1 + x_k z_k).$$

Here, instead of calculating inner product on `φ(x)` and `φ(z)` which would have involved `2^d` computations, we calculate inner products for `x` and `z` which require `d` computations. This kernel trick allows us to use all the advantages of higher dimensions, but by keeping the computation costs of our original space. Thus even though we learn our classifier in higher dimension, we never compute even once in this space.

Some common kernel functions are as follows:

Table 1. Some common kernel functions

Kernel	$K(x, x_j) =$	Kernel	$K(x, x_j) =$
Lineal	$x^T x_j$	Powered	$-\|x - x_j\|^\beta \; 0 < \beta \leq 1$
Polynomial	$(a \times x^T x_j + b)^d$	Log	$-log\left(1 + \|x - x_j\|^\beta\right) \; 0 < \beta \leq 1$
RBF	$e^{(-\frac{\|x - x_j\|^2}{\sigma^2})}$	Generalized Gaussian	$e^{-(x-x_j)^T A (x-x_j)}$ where A is a symmetric PD matrix
Sigmoid	$\tanh(\sigma x^T x_j + r)$	Hybrid	$e^{-\frac{\|x-x_j\|^2}{\sigma^2}} \times (\tau + x^T x_j)^d$

A linear kernel performs same as the linear model, but instead of storing `W`, now we store `aplha`. When there are lots of dimensions but not many training examples, using a linear kernel can be faster and more efficient. For eg. Datasets involving DNA or genetics.

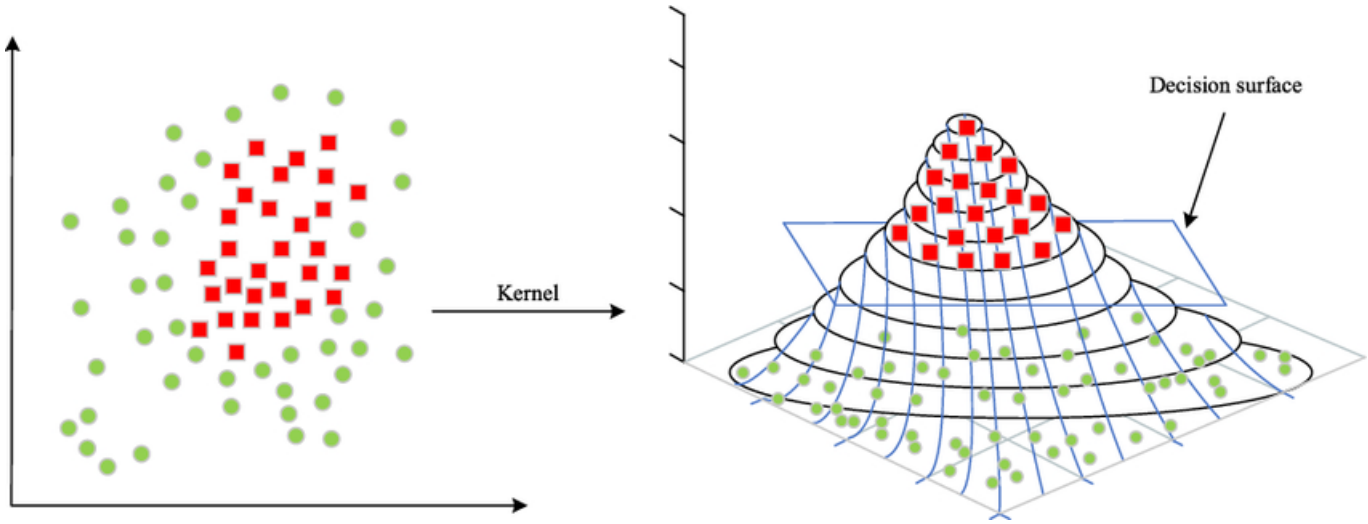
Now, the Radial Basis Function (RBF) kernel works differently. It puts Gaussian (bell-shaped) curves around each data point and finds the closest points based on these curves. Think of it like how K-Nearest Neighbors (KNN) works, but instead of using a fixed number of neighbors, the RBF kernel adjusts the number of neighbors based on these Gaussian curves. This makes it smarter than KNN because it adapts to the data better. It's interesting to see that when we try to make a linear classifier understand more complex relationships, we end up with ideas which are similar to KNN.

Algorithm

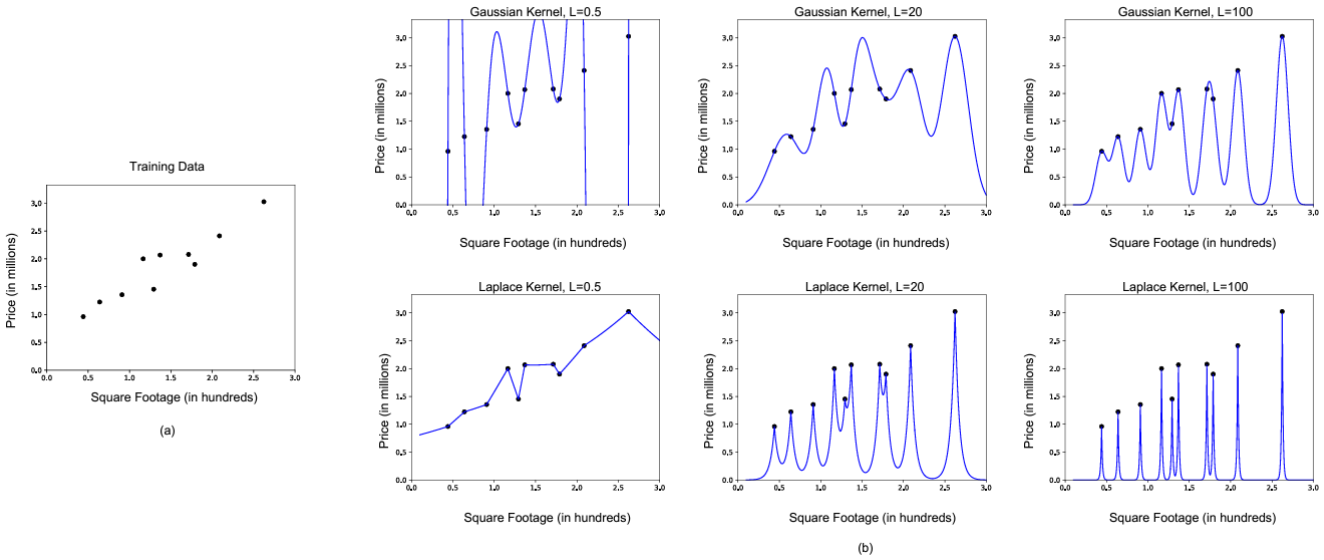
Any linear classifier can be kernelized using two steps.

- Step I: We need to show that the the classifier can be expressed solely in terms of inner products of feature vector.

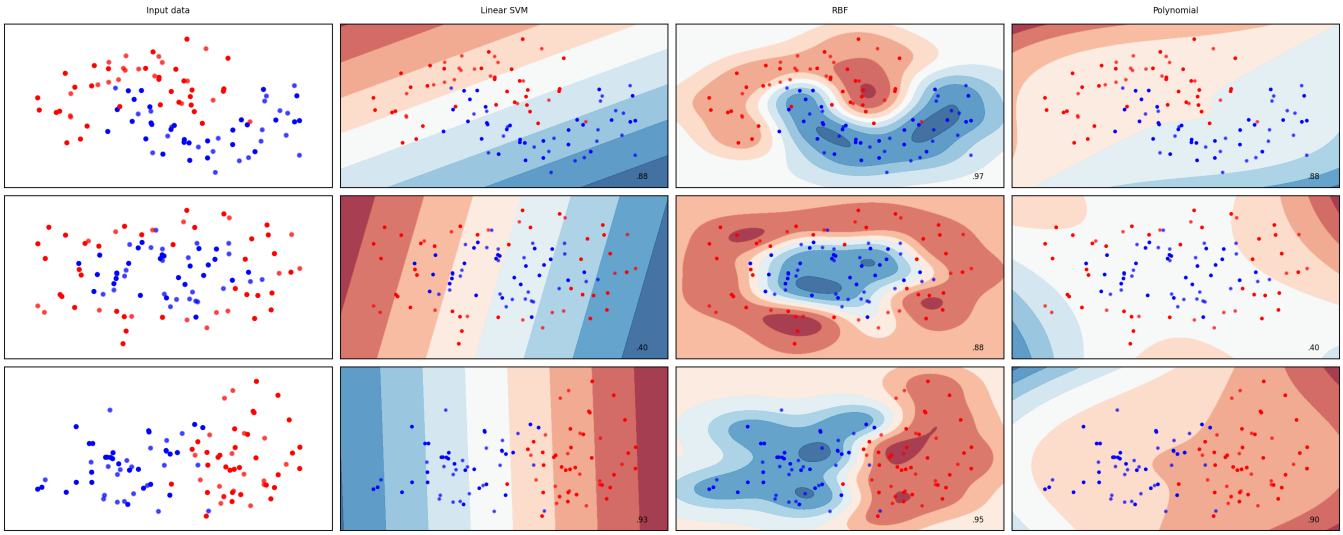
Kernelizing Perceptron



Kernelizing Linear Regression



Kernelizing SVMs



Results

One way to debug kernelized implementations is to use a linear kernel and ensure that the results match those obtained without using kernel methods. For instance, if include trigrams as feature and apply a linear kernel in kernel perceptron, we get exactly same results as those obtained without using kernels.

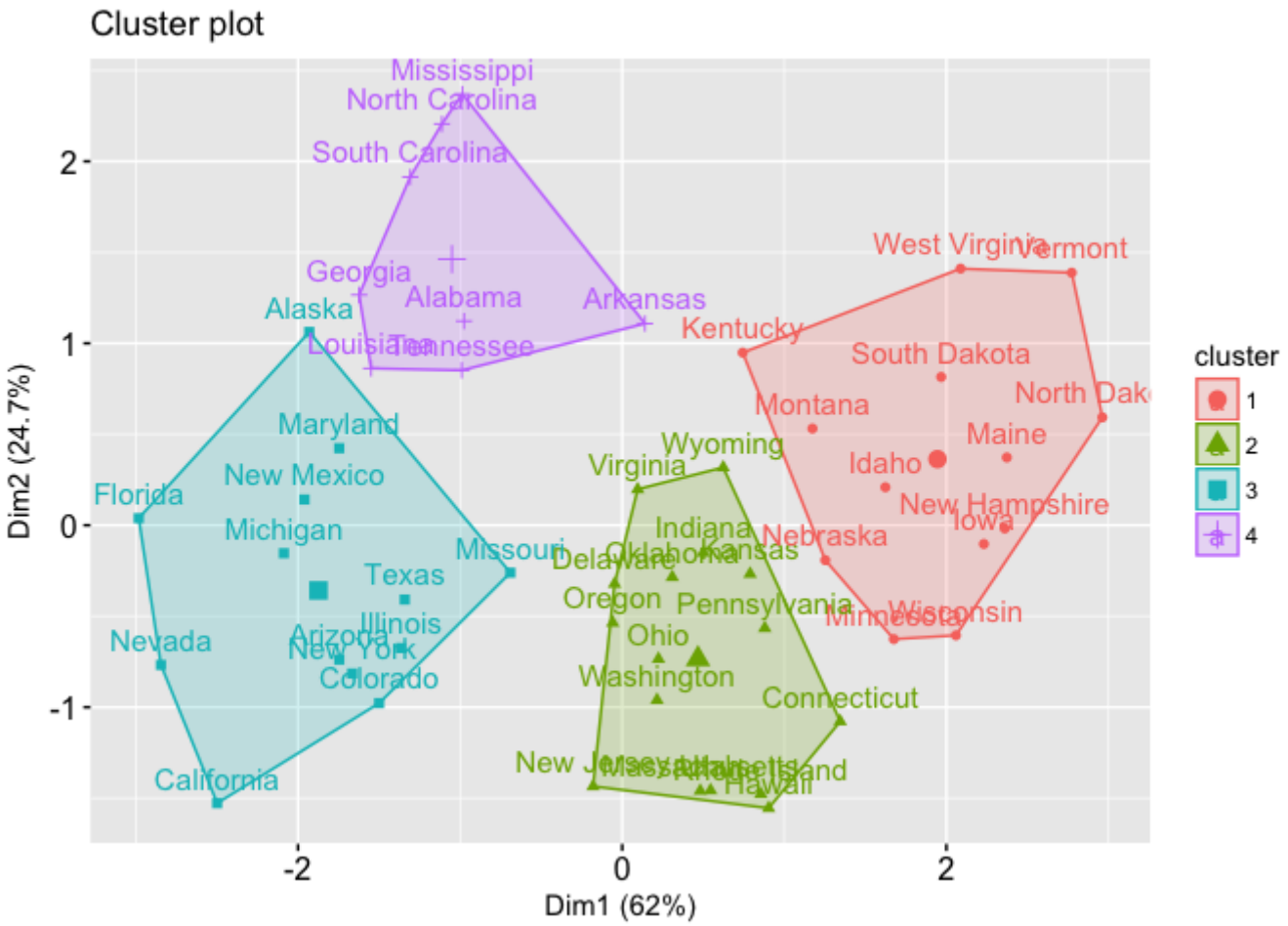
Perceptron

The Kernelized Perceptron converges in 12 steps when we consider features such as last character and bigrams from a name. Unlike regular Perceptron Algorithm whih required trigrams as a feature, Kernelized Perceptron are able to converge by just using last character and bigrams. It achieves an accuracy of `90%` on the training set which is significantly higher than `86.15%` by regular Perceptron. We use the Radial Basis Function Kernel for the same.

Linear Regression

Support Vector Machine

K Means Clustering



Introduction

Unsupervised Learning

Unsupervised learning is a method in machine learning where algorithms are employed to identify patterns in data without the need for labels. In simpler terms, it works with unlabeled data. The main aim of such methods is to uncover similarities, differences, or the underlying structure of the data.

Clustering

Clustering is a technique within unsupervised machine learning that focuses on grouping similar data points. The idea is to partition the data into distinct and exhaustive groups, or "clusters."

Clustering can be posed as a Constrained Optimization problem, where we try to find a C such that it leads to minimum dissimilarity. This is subject to the constraint that we can have at most K clusters.

$$variability(c) = \sum_{e \in c} distance(mean(c), e)^2$$

$$dissimilarity(C) = \sum_{c \in C} variability(c)$$

This is called as K-means objective function. Hierarchical clustering and K-means clustering are algorithms used to find locally optimum solutions to the clustering problem.

Hierarchical vs K-Means Clustering

Hierarchical clustering builds a hierarchy of clusters by either starting with individual data points as clusters and iteratively merging them or starting with all data points in one cluster and iteratively splitting them. This results in a tree-like structure called a dendrogram. Hierarchical clustering doesn't require specifying the number of clusters beforehand, which can be advantageous in certain scenarios. However, it can be computationally expensive for large datasets.

On the other hand, K-means clustering is an iterative algorithm that partitions the data into a pre-defined number of clusters (K) by minimizing the sum of squared distances between data points and their respective cluster centroids. K-means is computationally efficient and works well with large datasets, but it requires specifying the number of clusters beforehand and is sensitive to the initial choice of centroids.

Clustering vs Classification

The classification relies on labeled data for training. It learns from these labeled examples and then categorizes new instances based on what it has learned. In contrast, clustering works with unlabeled data. The clusters it generates represent partitions of the data, but these clusters don't carry semantic meaning as labels in classification do. Instead, they simply indicate similarities or patterns in the data.

Assumptions

Similar to K-nearest neighbors (KNN), K-means clustering operates under the assumption that instances close to each other in the feature space are likely to be similar. In essence, K-means seeks to partition the data into clusters where points within each cluster are more similar to each other than to points in other clusters. This assumption is fundamental to the algorithm's effectiveness.

Additionally, K-means assumes that the variance of the distribution of each attribute (variable) is spherical. This means that the clusters formed by K-means tend to be globular or spherical, and the algorithm performs optimally when the clusters have similar sizes and densities.

David Robinson's article K-means clustering is not a free lunch provides an intuitive explanation of why these assumptions are crucial.

Algorithm

```
randomly chose k examples as initial centroids
while true:
    create k clusters by assigning each
        example to closest centroid
    compute k new centroids by averaging
        examples in each cluster
    if centroids don't change:
        break
```

Initialization

The initial selection of centroids in K-means clustering greatly influences the outcome. If we start with centroids poorly positioned, the algorithm may converge to a suboptimal solution. One strategy to mitigate this is to initialize centroids randomly and run the algorithm multiple times, selecting the best solution among them. This increases the likelihood of finding a globally optimal clustering arrangement.

Choosing K

The parameter K in clustering represents how many groups we want to divide our data into. We can sometimes figure out K if we know our data well, but often we use methods like the elbow method.

The elbow method involves trying different K values and plotting how compact the clusters are. We look for where adding more clusters doesn't make much difference, forming an "elbow" shape in the plot. That's usually a good K value.

Results

Given a dataset of student marksheets, our task is to group them based on their features, such that similar students end up in similar groups. For a total of 75 groups, we performed clustering based on age, section, gender, and marks obtained in 4 subjects.

```
Group 0:
[16 'Gilberta' 'Male' 15 'A' 53 30 90 64]
[28 'Hanan' 'Male' 14 'A' 60 36 86 87]
[96 'Georgia' 'Female' 15 'A' 58 10 99 44]
Centroid: [ 0.7  0.8  0.  57.  25.3 91.7 65. ]

Group 9:
[11 'Dunn' 'Male' 15 'C' 100 93 87 81]
[179 'Val' 'Male' 13 'B' 95 73 91 61]
Centroid: [ 1.   0.5  1.5 97.5 83.  89.  71. ]
```

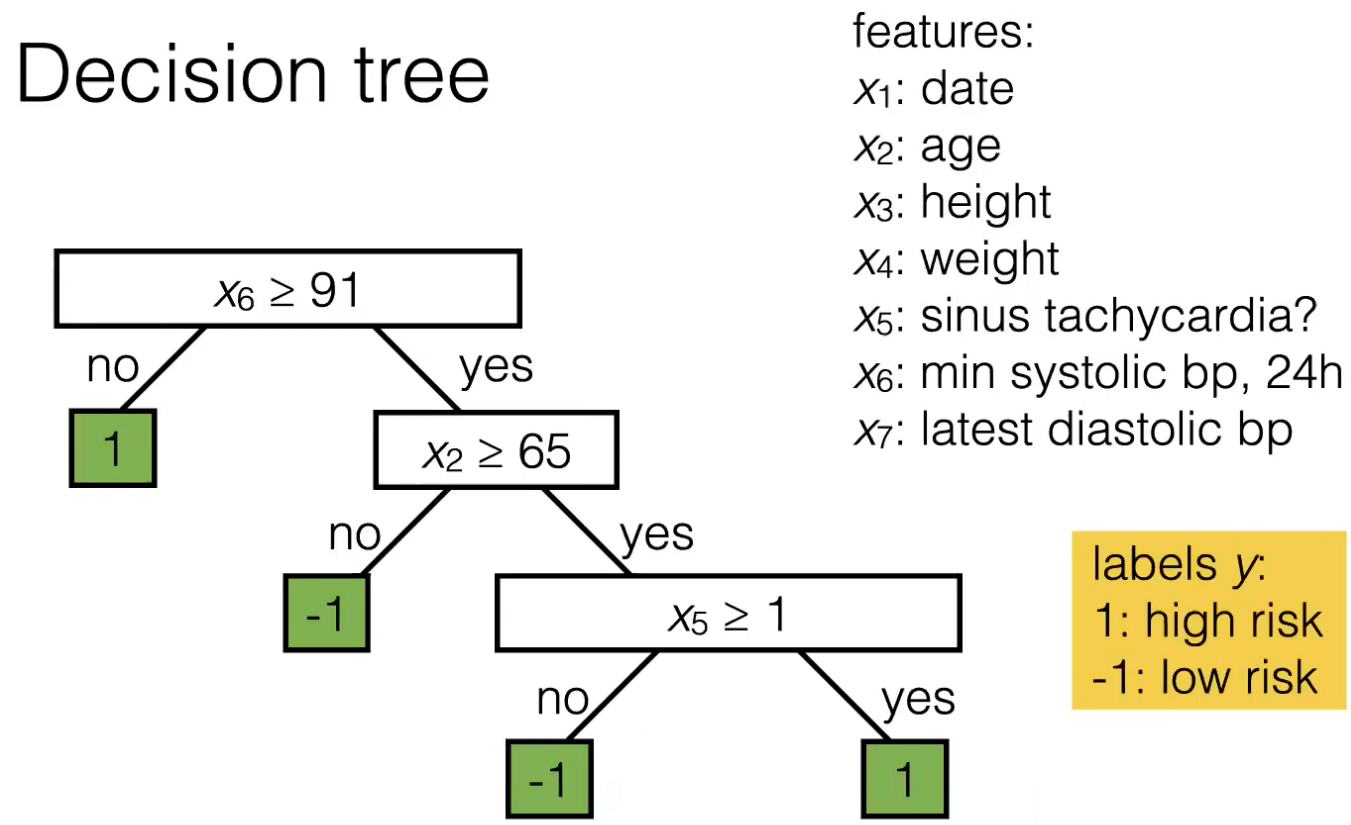
Here we can observe that there is a decent consistency among all groups. Also by modifying the distance function, we can group students based on specific parameters. Considering only maths and science

performance, we obtain the following results:

```
Group 0:
[73 'Lorrie' 'Male' 13 'A' 72 78 40 78]
[102 'Alvin' 'Female' 13 'B' 80 28 53 67]
[120 'Alyse' 'Male' 14 'A' 80 34 85 71]
[125 'Fredric' 'Male' 14 'B' 68 20 21 81]
[145 'Alis' 'Male' 15 'A' 76 53 32 69]
[158 'Aldin' 'Female' 13 'C' 72 24 83 62]
[185 'Purcell' 'Male' 14 'B' 75 14 54 68]
Centroid: [ 0.7  0.4  0.7 74.7 35.9 52.6 70.9]

Group 9:
[43 'Brandie' 'Male' 15 'B' 43 2 17 78]
[84 'Eugine' 'Male' 13 'C' 34 18 39 94]
[98 'Silvia' 'Male' 14 'A' 40 61 63 80]
[104 'Woodie' 'Male' 15 'A' 37 57 48 75]
[147 'Jaquenette' 'Male' 15 'C' 33 75 98 84]
[155 'Emlyn' 'Male' 14 'C' 32 3 69 96]
[171 'Fran' 'Female' 15 'B' 34 74 83 81]
Centroid: [ 0.9  0.7  1.1 36.1 41.4 59.6 84. ]
```

Group 0 consists of all students with great performance in both maths and science, while group 9 consists of students with great performance in Maths and average performance in Science.

Decision Trees



Introduction

Decision Trees are a supervised learning algorithm used for both classification and regression tasks. They represent decisions using a series of if-then-else rules, which can be visualized as a tree-like structure.

Intuition

Let's say we use K-Nearest Neighbors for binary classification. KNN can be slow, but we can speed it up using KD Trees.

A KD Tree is a binary tree that helps speed up nearest neighbor searches by partitioning data along its axes. When searching for the nearest neighbor, the algorithm recursively checks nodes, comparing distances to find the closest points. At each step, we compare the current best distance with the distance to the opposite branch. If the search radius (current best distance) intersects the splitting plane, we continue searching that branch, because we want to find the exact nearest neighbor.

However, we can make it faster by not searching for the exact nearest neighbor, but rather approximately nearest neighbor. This can be done by not performing recursion. While this idea makes inference faster, it loses upon the accuracy slightly. We can make it even faster by introducing the concept of purity. A "pure" node is a node where all points belong to the same class. If we encounter such a node during a decision, we can immediately make a prediction without further checks. This greatly speeds up the process and reduces the amount of data stored, as we only need the structure of the tree.

Thus, Decision Trees are essentially KD Trees, but the only difference being splitting criteria. In KD Trees, we split data to get two equal halves. However, in Decision trees, we split data to get pure nodes.

Bias and Variance are a function of depth of tree. As the depth of tree increases, bias decreases and variance increases.

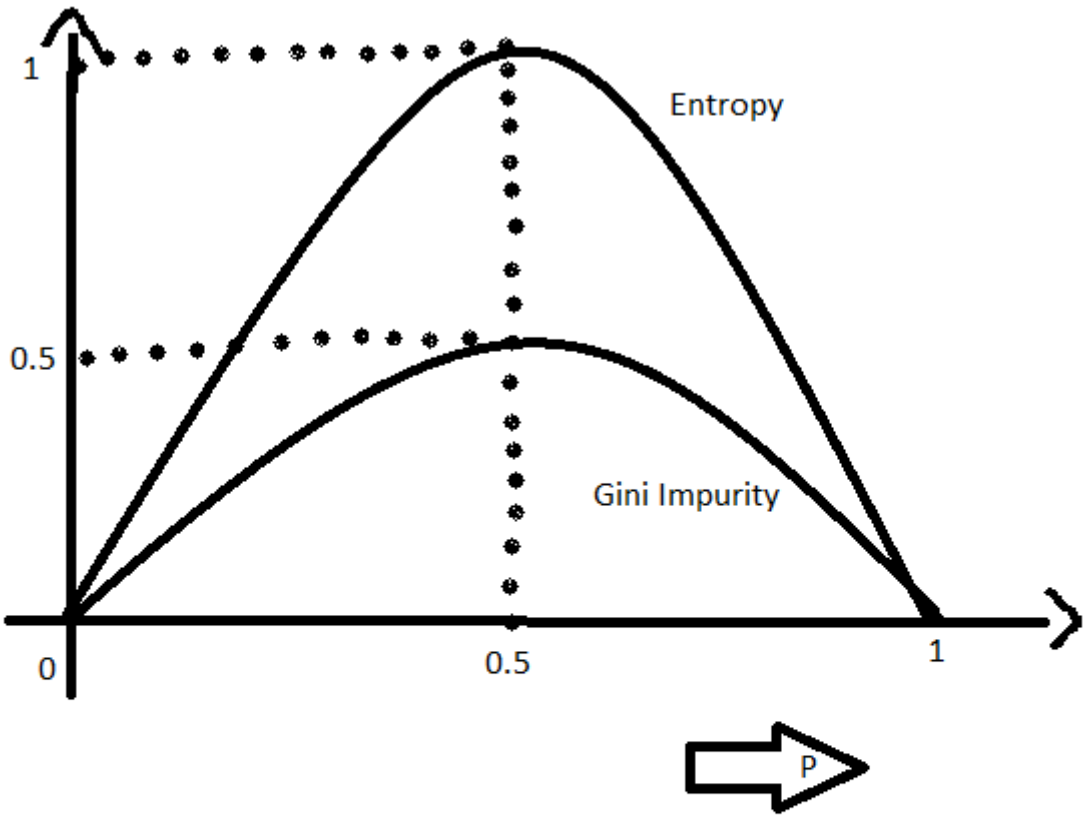
Algorithm

Decision Trees are a non-parametric algorithm. This means they don't have fixed parameters (like weights W and bias b in a perceptron). Instead, each non-leaf node in the tree is defined by two things:

1. Split dimension: The feature (or attribute) of the data used to make a decision at this node.
2. Split value: The value that the feature is compared against to determine which branch to follow.

One way to find these parameters is to try every split dimension and every split value, and then evaluate it using a impurity function.

Impurity Functions

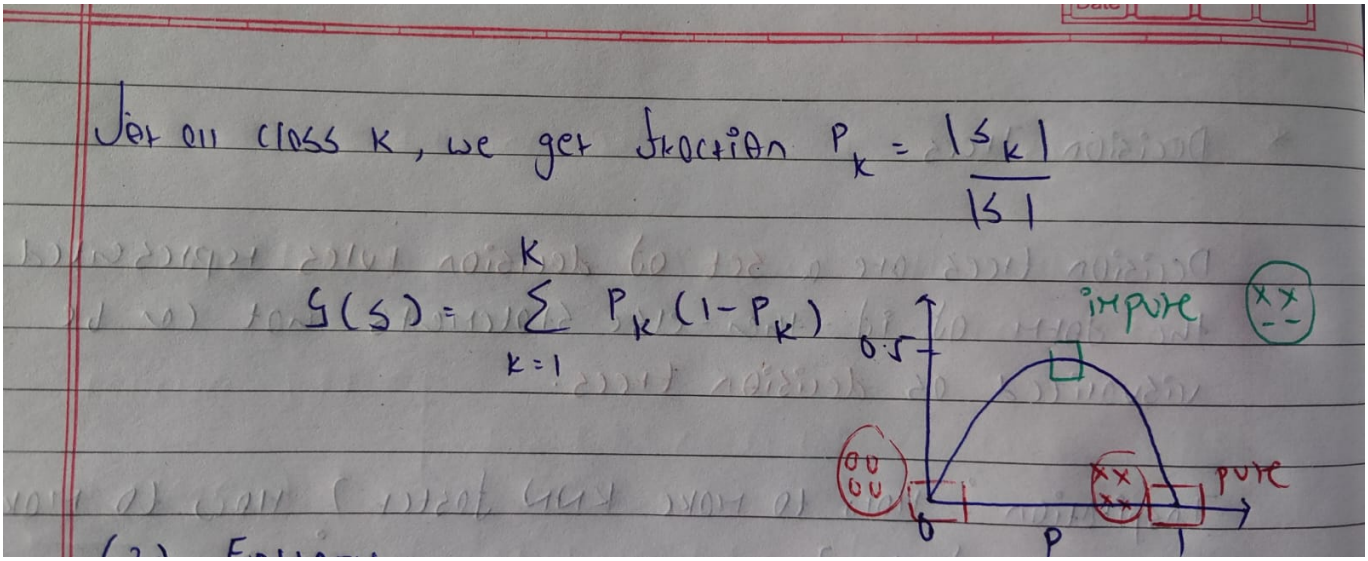


$$E(S) = \sum_{i=1}^{c} - p_i \log_2 p_i$$

$$Gini(E) = 1 - \sum_{j=1}^{c} p_j^2$$

Gini Index

Consider S as the dataset available at node N. We can compute the Gini Index of this node using the below formula.



Entropy

Entropy for a subset `S`, where proportion of positive and negative examples are given by `p(+)` and `p(-)` respectively is given by,

$$H(S) = - p_{(+)} \log_2 p_{(+)} - p_{(-)} \log_2 p_{(-)}$$

One way to look at entropy is that it represents the total number of bits required in order to express the class of a random instance of subset `S`. In other words, if we were to pick a random sample from `S`, what is the total number of bits required to express its class.

Consider a pure subset which consists only of positive elements. In this case, entropy = H(S) = 0. Thus, we dont require any bits to determine the class of the instance. We can be sure its a positive instance becuase the subset is positive.

Now, consider an impure subset such that there are 3 positives and 3 negatives. Here, we get `H(S) = 1`. This means that we require one full bit to determine the class of instance.

All this says is how uncertain the subset S is.

An explaination for this intuition can be found here

(2) Entropy.

Consider a distribution Q, such that for K classes,

$$q_1 = q_2 = q_3 = \cdots = q_K = \frac{1}{K}$$

This means that all points are uniformly distributed. This is a distribution, that we do NOT want. Here, we want a distribution P which completely diverges with Q.

Thus, we want to Maximize the KL divergence

$$KL(P||Q) = \sum_{k=1}^{K} p_k \log_2\left(\frac{p_k}{q_k}\right).$$

$$\therefore \text{Max}\left[KL(P||Q)\right] = \text{Max}\left[\sum_{k=1}^{K} p_k \log_2\left(\frac{p_k}{q_k}\right)\right]$$

But, we know $q_k = \frac{1}{K}$.

$$\therefore \text{Max}\left[\sum_{k=1}^{K} p_k \log_2\left(\frac{p_k}{(1/K)}\right)\right]$$

$$= \text{Max}\left[\sum_{k=1}^{K} p_k \log_2 p_k + \sum_{k=1}^{K} p_k \log_2 K\right]$$

$$= \text{Max}\left[\sum_{k=1}^{K} p_k \log_2 p_k + \underbrace{\log_2 K}_{\text{constant}} \underbrace{\left(\sum_{k=1}^{K} p_k\right)}_{\to 1}\right]$$

$$= \text{Max}\left[\sum_{k=1}^{K} p_k \log_2 p_k\right] = \text{Min}\left[-\sum_{k=1}^{K} p_k \log_2 p_k\right]$$

Regression Trees

The only difference in regression trees is that we use a loss function like MSE instead of impurity functions with respect to mean of the points in that subset.

Training a decision tree

Training a decision tree is different from other ML models because it doesn't require optimization methods like gradient descent. It reduces the loss by decreasing impurity. Also the regularizer here is the depth of tree or number of nodes. Minimizing the purity is a NP hard problem, and hence we choose a greedy way to approach it.

Building a decision tree

- Regression tree with squared error loss

BuildTree(I; k)

if $|I| \leq k$

Set $\hat{y} = \text{average}_{i \in I} y^{(i)}$

return Leaf(label = $\hat{y}$)

else

for each split dim j & value s

Set $I^+_{j,s} = \{i \in I | x^{(i)}_j \geq s\}$

Set $I^-_{j,s} = \{i \in I | x^{(i)}_j < s\}$

Set $\hat{y}^+_{j,s} = \text{average}_{i \in I^+_{j,s}} y^{(i)}$

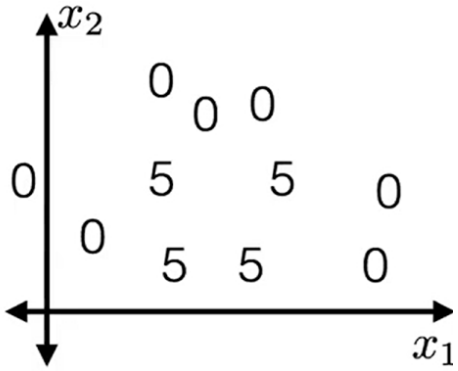
Set $\hat{y}^-_{j,s} = \text{average}_{i \in I^-_{j,s}} y^{(i)}$

Set $E_{j,s} = \sum_{i \in I^+_{j,s}} (y^{(i)} - \hat{y}^+_{j,s})^2 + \sum_{i \in I^-_{j,s}} (y^{(i)} - \hat{y}^-_{j,s})^2$

Set $(j^*, s^*) = \arg\min_{j,s} E_{j,s}$

return Node

8



BuildTree({1,…,n};2)