BI-VZD notes

February 18, 2020

1 Decision trees

1.1 Construction

The **ID3** algorithm by John Ross Quinlan is commonly used to construct decision trees. It picks a feature that most nicely divides the input data according to a metric. This metric must have the following properties:

- It should be non-negative
- It should be equal to zero if the input set is a single equivalence class
- It should reach a maximum if the ratios of individual elements in the input data are equal
- It should be increasing on $[0, \frac{1}{2}]$ and decreasing on $[\frac{1}{2}, 1]$

Such a function measures disorder, it is called **entropy**. For binary input:

$$H(D) = -p_0 \log p_0 - p_1 \log p_1 = -p_0 \log p_0 - (1 - p_0) \log(1 - p_0)$$

In general:

$$H(D) = -\sum_{i=0}^{k-1} p_i \log p_i$$

When choosing a feature to construct a decision tree, it is desirable to decrease disorder as much as possible for the child nodes. This decrease is called **information gain** and is defined as "entropy of D minus the weighted sum of entropies D_0 and D_1." Formally:

$$IG(D, X_i) = H(D) - t_0 H(D_0) - t_1 H(D_1)$$

where D_0 and D_1 are subsets of D and correspond to the data to be partitioned in the child nodes, for which $X_i = 0$ and $X_i = 1$, respectively. Also, $t_i = \frac{|D_i|}{|D|}$.

Aside from entropy, a common choice for a metric of "the probability of misclassification of a new data point" is the **Gini impurity**:

$$GI(D) = 1 - \sum_{0}^{k-1} p_i^2 = \sum_{0}^{k-1} p_i(1 - p_i)$$

Construction of decision trees is an NP-complete problem, which is why greedy algorithms are used in practice. These often find a suboptimal solution. They typically advance recursively; for a given dataset, a feature is found which partitions the set in a way which maximizes information gain or Gini index. The same method is then applied on the subsets resulting from the partition. This process is terminated by a **termination condition**, such as maximum tree depth, minimal subset size, minimal information gain, etc.

1.2 Training and testing sets

Regarding classification, for now a notion of **classification accuracy** will be sufficient. It is simply

$$\frac{\text{the number of correctly classified data points}}{\text{size of the entire dataset}}$$

A better approximation of model accuracy is arrived at by splitting the dataset into two parts, the so-called train set and test set. The error (a measure of inaccuracy) on these sets is simply called train error and test error. This approach can help prevent **overfitting**, which is the tendency of the model to fit the training data too specifically, rendering it useless when faced with unseen data points.

1.3 Hyperparameter tuning

One approach is to

1. split the dataset into training and testing sets
2. for different tree depths, note the classification accuracy on the testing data
3. choose the depth with the lowest test error
4. assume this error to be an estimate of the error on real data, i.e. an objective measure of quality of the model

There is a problem with this approach, however. The test error will typically be an overly optimistic estimate of real data. The tree depth was chosen based on a limited input dataset and is therefore adapted to this specific set. In other words, this approach violates the rule of not touching testing data while designing and training the model.

This issue can be resolved by splitting the original dataset into three, not two, subsets. The new subset is called the **validation set**, and exists solely for the purpose of estimating the error on real data.

1.4 Multivalued features

Decision tree implementations typically only support binary target variables and binary features. To use decision trees on features which have more than two values, these features are encoded with **one-hot encoding**. This encoding method simply replaces a feature with n distinct values with n binary **dummy variables**. It has several downsides:

- significantly increases the number of features, which may lead to overfitting
- while it doesn't remove information from the dataset, greedy algorithms may still decrease the quality of the model
- it is not suitable for all kinds of categorical features, especially if there's an order to the categories[1] involved.

There are two kinds of categorical features:

- **Nominal** There's no order to the categories, e.g. place of birth, faculty, gender, etc.
- **Ordinal** The categories are ordered, e.g. education (elementary $<$ high school $<$ university) etc.

One-hot encoding is therefore best used for nominal features only. Ordinal features can be interpreted as continuous features instead.

1.5 Continuous features

In decision trees, continuous features can also be used for branching (e.g. age $<$ 30). Unlike discrete features, continuous ones can appear in multiple levels of the tree. This necessarily implies changes to the decision tree construction algorithm, which needs to take the continuous features into account when choosing a branching condition. This is done as follows:

1. The possible values of the continuous feature X in the to-be-partitioned set D are ordered into a sequence $x_1 < x_2 < \cdots < x_l$
2. The different partitionings of D $(\forall i \leq l)$ according to the condition $X < x_i$ carried out and their respective information gains are computed
3. The best partitioning is the one with the highest information gain

If there are too many possible values of X in D, then only some subset is considered (often a random one).

1.6 Decision trees for regression

For use of decision trees to model continuous target variables, two problems have to be resolved:

1. How will a trained decision tree determine the predicted value of the target variable?
    - For discrete target variables, this was simply the most common value in a leaf of the decision tree.
    - For continuous variables, it's possible not to have two equal values in the dataset (generally the possibility of two values being equal is infinitely lower).

[1]the categorical theorist in me cries in pain upon witnessing the abuse of the word "category" for such lame theory.

2. What criterion should be used for the choice of the best feature for branching?
    - For discrete variables, entropy or Gini index can be used. These are both unusable for continuous variables (the probability of any single value is zero)

The first problem is usually resolved by setting the target variable to the *average* of the values in the leaf of the tree.

The second problem requires a little more work. Since the leaf values are averaged, the tree should try to bring the values in each node as close to the expected value as possible. A well-known measure of deviation from the expected value is **MSE = Mean Squared Error**. It is almost equal to **sample variance**.

$$MSE(\boldsymbol{Y}) = \frac{1}{N}\sum_{j=0}^{N-1}(Y_i - \overline{Y})^2$$

1.6.1 CART algorithm

The greedy algorithm, in which nodes of the decision tree are chosen so as to minimize MSE is called **CART = Classification And Regression Trees**. Its general structure is identical to that of ID3; it also has a notion of information gain, computed as

$$MSE(D) - t_L MSE(D_L) - t_R MSE(D_R)$$

where $t_L = \frac{|D_L|}{|D|}$ (analogically for t_R). Instead of MSE, MAE = Mean Absolute Error is sometimes used:

$$MAE(\boldsymbol{Y}) = \frac{1}{N}\sum_{j=0}^{N-1}|(Y_i - \overline{Y})|$$

Decision trees come with many pros:

- They don't require much preprocessing, supporting categorical and continuous variables alike. They are also resilient to missing data.
- They are easily understandable, their training is relatively fast
- They are easy to interpret, their decisions are trivial to explain

But they have their cons as well:

- They aren't *robust*: a small change in training data can significantly restructure the tree
- Most implementations only support binary trees
- Finding an optimal tree is an NP-complete problem
- It is very easy to overfit them

2 Ensemble methods

The basic principle of ensemble methods is that rather than using a single model, we use multiple models and combine their results. The two most common ensemble methods are **bagging = bootstrap aggregating** and **boosting**. Examples of these approaches include **random forests** and **AdaBoost** (Adaptive Boosting).

2.1 Random forests (bagging)

1. The input dataset D is split into n datasets D_i of the same size by picking data from D and repeating it (this is the bootstrapping part).
2. A decision tree T_i is trained on each dataset D_i. It can be very shallow, often a depth of 2 or 3 is used, but even 1 is a valid option.
3. Each data point (a row from D) is fed to all trees $T_1, \dots, T_n$ and from each tree T_i a decision Y_i is fetched.
4. The trees $T_1, \dots, T_n$ are a **random forest**, its final decision is determined by a majority vote on $Y_1, \dots, Y_n$ for discrete variables, or by the average of the individual decisions for continuous variables.

It is very important that the models combined with ensemble methods are as varied as possible. This is usually achieved by randomizing some (hyper)parameters of the models. In the case of random forests, this randomness is provided by the choice of data points in the individual datasets D_i and by the hyperparameter `max_features`, which limits the number of (randomly) chosen features from which the greedy algorithm picks a feature for branching. Since decision trees are so sensitive to changes in the training set, bootstrapping is often enough to produce highly varied decision trees.

Even though individual decision trees can be very weak models (submodels in ensemble methods are called *weak learners*), their collective decisions can lead to surprisingly good results. Unlike individual decision trees, random forests are very robust and quite resilient to overfitting. These improvements come at the expense of simplicity and the quality of being easy to interpret.

2.2 AdaBoost (boosting)

Boosting makes use of **sample weights**, which are a sequence of N real numbers which add up to 1 (N is the number of data points – samples – in D). For decision trees, sample weights are applied when computing the information gain/Gini index of a partition. Recall

$$IG(D, X_i) = H(D) - t_0 H(D_0) - t_1 H(D_1)$$

Sample weights determine t_0 and t_1 in this equation: the coefficients are sums of the weights of samples in D_0 and D_1, respectively. Using sample weights has the result of prioritizing correct prediction of samples with a higher weight.

Like in the case of bagging, boosting uses multiple submodels. The final decision is the weighted sum of the individual submodels' decisions. Unlike bagging, however, the submodels employed by boosting are *ordered* and every model is affected by the ones which come before it. This effect is realized using sample weights: when constructing the n-th tree, the weights of all samples misclassified by the $(n-1)$-th tree are increased. These weight changes ensure that each added model focuses on classifying the samples which previous models could not handle. This is roughly how boosting works in general, what follows is specific to AdaBoost.

2.2.1 AdaBoost

The algorithm begins with a dataset D with N data points (samples). To simplify matters a bit, let us only consider binary classification for this example. The number of constructed trees is controlled by the hyperparameter `n_estimators`.

1. The initial weights are set uniformly, i.e. $w_i = \frac{1}{N}$ and $m = 1$.
2. If $m \leq$ `n_estimators`, construct a tree $T^{(m)}$ on dataset D with weights w_i.
3. Let $e^{(m)}$ be the sum of weights of samples in D misclassified by $T^{(m)}$.
4. If $e^{(m)} = 0$, terminate the algorithm. All data is classified correctly.
5. Let
$$\alpha^{(m)} = \texttt{learning\_rate} * \log \frac{1 - e^{(m)}}{e^{(m)}}$$
where `learning_rate` is a hyperparameter (set by the user) used to slow learning down in order to prevent overfitting (if it is less than one).
6. For samples misclassified by $T^{(m)}$, set new weights $w_i \leftarrow w_i \exp(\alpha^{(m)})$.
7. Normalize weights so that $\sum_{i=0}^{N-1} w_i = 1$.
8. $m \leftarrow m + 1$, go to 2.

To determine the decision of an AdaBoost model for a sample x:

1. To every tree $T^{(m)}$, a weight determined by $\alpha^{(m)}$ is assigned.
2. The weights $\alpha^{(m)}$ are summed up for all trees predicting $Y = 1$ for x. The same is done for all trees predicting $Y = 0$.
3. Choose the option with the highest sum of weights as the final decision.

There exist variants of AdaBoost for classification of non-binary variables (*AdaBoost-SAMME*) and regression (*AdaBoost.R2*). AdaBoost doesn't necessarily rely on decision trees, the submodels can be of any kind of model which supports `sample_weight`.

The `learning_rate` hyperparameter is an example of **regularization**: the lower it is, the more resilient the model is to overfitting. The downside is that lower learning rates typically require more submodels (`n_estimators`).

3 Unsupervised learning

In situations where the dataset isn't tagged in any way (there's no metric to predict), we talk about unsupervised learning. Its goal is to understand the structure of the input dataset. This understanding typically means recognition of the local properties of subsets of the dataset.

Consider a situation where the entry dataset contains p features. Let χ be the space in which possible results can be found[2].

- For n binary features, let $\chi = \{0, 1\}^p$.
- For n continuous features, let $\chi = \mathbb{R}^p$.

From the probability theory perspective, we interpret the entry dataset as *realizations of the random vector* $\boldsymbol{X} = (X_1, \ldots, X_p)^T$. Understanding the internal structure means understanding the partition $\boldsymbol{X}$. We would like to *estimate the probability* $P(\boldsymbol{X} \in O)$ for every subset $O \subset \chi$.

In the case of continuous features, this is equivalent to estimating the compound *probability function* $p_{\boldsymbol{X}}(\boldsymbol{x}) \equiv P(X_1 = x_1, \ldots, X_p = x_p)$

We focus first and foremost on finding the areas in χ which have a high probability but at the same time are as "small" as possible.

3.1 Cluster analysis

The goal is to classify the data into *clusters*, so that these two conditions are met (to some degree):

- Points which are close to one another will belong into the same cluster and
- points which are far away from one another will belong to different clusters.

This description is vague at best and its formalization is unclear. What makes matters worse is the absence of a criterion which would determine the quality of a specific clustering.

A key concept in clustering is **distance**.

Distance on a set χ is a function $d : \chi \times \chi \longrightarrow [0, +\infty)$ such that $\forall x, y, z \in \chi$

1. $d(x, y) \geq 0$ and $d(x, y) = 0 \iff x = y$ (positive definiteness)
2. $d(x, y) = d(y, x)$ (symmetry)
3. $d(x, y) \leq d(x, z) + d(y, z)$ (triangle inequality)

Metric space is the pair (χ, d).

The following functions are a common choice for the distance metric:

1. *Euclidean distance* (also called L_2 distance)

$$d_2(x, y) = \sqrt{\sum_{i=1}^{p} (x_i - y_i)^2}$$

[2]The choice of χ is non-trivial, it is one part of the process of choosing a model.

2. *Manhattan distance* (also called L_1)

$$d_1(x, y) = \sum_{i=1}^{p} |x_i - y_i|$$

3. *Chebyshev distance* (L_∞)

$$d_\infty(x, y) = \max_i |x_i - y_i|$$

3.1.1 Inputs and outputs of clustering

- Inputs
  1. Metric space χ with distance d
  2. Dataset $D \subset \chi$
  3. The desired number of clusters k
- Outputs
  1. The decomposition of D into individual clusters $C_1, \ldots, C_k$
  2. For hierarchical clustering, an additional output can be a *dendrogram*, a graphic representation of the hierarchical structure of clustering.

3.2 Hierarchical clustering

Let $N = |D|$.

- Let each data point be a cluster, i.e. let there be N clusters.
- Repeat the following steps:
  - Find two clusters closest to each other
  - Unify these two clusters, decrementing the current number of clusters

After $N - 1$ steps, all samples belong to a single cluster. For this process to take place, a notion of cluster distance has to be established. That is, the distance function has to be extended to support entire clusters as inputs. Additionally, to produce a useful result, a terminating condition has to be set (this is usually k or a minimal distance between clusters).

3.2.1 Cluster distance

Let A, B be clusters. Then the function $D(A, B)$ is a distance metric between clusters, usually chosen from one of the following implementations:

- *Single linkage* - generates long chains

$$D(A, B) = \min_{x \in A, y \in B} d(x, y)$$

- *Complete linkage* - generates compact clusters

$$D(A, B) = \max_{x \in A, y \in B} d(x, y)$$

- *Average linkage* - a compromise between the two

$$D(A,B) = \frac{1}{|A||B|} \sum_{x \in A, y \in B} d(x,y)$$

- *Ward's method* in $\mathbb{R}^p$, very effective, minimizes inner variance

$$D(A,B) = \sum_{x \in A \cup B} ||x - \overline{x}_{A \cup B}||^2 - \sum_{x \in A} ||x - \overline{x}_A||^2 - \sum_{y \in B} ||y - \overline{y}_B||^2$$

where $\overline{x}_A = \frac{1}{|A|} \sum_{x \in A} x$ is the barycenter (centroid) of set A, analogically for $\overline{x}_{A \cup B}$ and $\overline{y}_B$.

3.2.2 Visualization

The process of agglomerative clustering can be visualized with a *dendrogram*. It is a tree, its nodes represent clusters formed during the algorithm's operation. The leaves are the initial single-element clusters, the root is the final all-encompassing cluster.

3.3 k-means clustering

A common approach to clustering interprets the problem as an optimization task. For such an interpretation, an *objective function* is needed. This function assigns a certain decomposition into clusters a *score* (or fitness, etc.). The goal of clustering is then to minimize (or maximize, depending on definition) this function.

For a given k, we're looking for a decomposition of D into $C = (C_1, \ldots, C_k)$ in $\chi = \mathbb{R}^p$ equipped with Euclidean distance minimizing the objective function

$$G(C) = \sum_{i=1}^{k} \frac{1}{2|C_i|} \sum_{x,y \in C_i} d(x,y)^2 = \sum_{i=1}^{k} \frac{1}{2|C_i|} \sum_{x,y \in C_i} ||x-y||^2$$

The problem of finding a global optimum is NP-hard. However, an iterative heuristic-based algorithm can be used instead, which converges towards a local optimum.

3.3.1 The correspondence between an objective function and a centroid

Claim For a finite set (of points) $A \subset \mathbb{R}^p$

$$\frac{1}{2|A|} \sum_{x,y \in A} ||x-y||^2 = \sum_{x \in A} ||x - \overline{x}||^2 = \min_{\mu \in \mathbb{R}^p} \sum_{x \in A} ||x - \mu||^2$$

(For a proof of this claim, please follow the official handouts.)

Thanks to this claim, the objective function can be rewritten as

$$G(C) = \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \overline{x}_i||^2$$

where $\overline{x}_i$ is the centroid of i-th cluster. Let us now set $\mu_i = \overline{x}_i$. Now let $\widetilde{C} = (\widetilde{C}_1, \dots, \widetilde{C}_k)$ such that the point x is now located in the cluster $\widetilde{C}_i$ such that $||x - \mu_i||$ is the smallest possible. This action decreases the sum of squared distances,

$$\sum_{i=1}^{k} \sum_{x \in \widetilde{C}_i} ||x - \mu_i||^2 \leq \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||^2$$

The left hand side of this inequality can be rewritten according to the claim above as

$$G(\widetilde{C}) = \sum_{i=1}^{k} \sum_{x \in \widetilde{C}_i} ||x - \overline{\overline{x}}_i||^2.$$

Therefore, $G(\widetilde{C}) \leq G(C)$.

3.3.2 The k-means algorithm

Initialization phase: initial placement of the centroids $\mu_1, \dots, \mu_k$. *Iterative phase*:

1. Assign points into clusters: $C_i = \{x \in D | i = \arg\min_j ||x - \mu_j||\}$
2. Recompute points $\mu_1, \dots, \mu_k$ as centroids of these clusters:

$$\mu_i \leftarrow \frac{1}{|C_i|} \sum_{x \in C_i} x$$

The algorithm terminates once the change of the objective function becomes sufficiently small. The result of this algorithm can be altered significantly by the initialization phase. The initialization phase is often generated randomly. There are smarter ways to initialize k-means, however.

3.3.3 Choosing k

Unlike hierarchical clustering, k-means requires k to be set in advance. There is no universal approach to determining an optimal k. A decent heuristic is based on the following realization:

- Let k^* be the optimal number of clusters.
- Then for $k < k^*$, the objective function will decrease swiftly when k is increased. For $k \geq k^*$, the objective function will increase slower.
- A good approximation of k^* can therefore be achieved by plotting the objective function for various k and looking for *elbows* in the graph.

This heuristic is unfortunately highly subjective and practically unusable for certain datasets.

4 k Nearest Neighbors

In a supervised learning problem, there's a training dataset $X \in \mathbb{R}^{N,p}$ with known values of the target variable $Y \in \mathbb{R}^N$. The basic principle of kNN (k nearest neighbors) is that to determine the value of Y for a point x, we find k (k is a hyperparameter) points closest to x in the training dataset, and use their average (for continuous variables) or most common value (for discrete variables) as the final decision.

Recall the concept of a *distance metric*. The L_2 (Euclidean) distance is the most common choice for this metric.

With decision trees, the training phase was computationally expensive, but individual predictions were trivial to construct. This is typical for supervised learning. In the case of kNN, however, the opposite applies. **The training dataset is already a trained model**. It is the individual predictions which are computationally expensive. Predictions can be sped up by building an index (typically a search tree) during the learning phase.

4.1 Hyperparameters of kNN

- k – the number of neighbors to search for. A higher k can prevent overfitting.
- `metric` – the distance metric
- `weights` – the weights of the neighbors' predictions

4.1.1 L_k distance metrics

The L_1, L_2, and L_∞ previously mentioned metrics can be generalized as follows:

$$L_k = d_k(x, y) = \sqrt[k]{\sum_{i=0}^{p-1} |x_i - y_i|^k}$$

Distance metrics for kNN can be chosen arbitrarily, however. Complex distance metrics can turn simple kNN into complex (and powerful) models, with additional hyperparameters tweaking the metric itself.

4.1.2 The `weights` hyperparameter

A prediction can be determined by a simple average of the k neighbors, but usually a weighted average is chosen instead. The weights of neighbors x_i are often set to decrease with distance from the input sample x, i.e.

$$w_i = \frac{1}{d(x, x_i)}$$

4.2 Normalization and nominal features

Unlike decision trees, the kNN method usually requires preprocessing of the entry dataset and is sensitive to the types of individual features. For now, let's assume that all features are numerical and continuous.

A major problem with kNN is the effect units of features have on the values of the distance metric. For example, having one feature in centimeters and another in meters significantly prioritizes the latter one, as distances on that axis won't seem nearly as long as the same distances on an axis in centimeters, even though they are in fact equal. This particular case can be resolved very easily, but often there are features with significantly different scales which are essentially incomparable (unlike meters vs centimeters).

A common (but naive) solution to these problems is **data normalization**. This is a process of rescaling each feature on the interval $[0, 1]$, using the simple formula

$$x_i' = \frac{x_i - \min_x}{\max_x - \min_x}$$

When faced with features of similar or even identical scales (such as individual pixels in an image represented by their brightness), one has to consider the problems normalization may introduce. Such features are typically better rescaled using the minima and maxima across the entire set of related features, otherwise, information about possible dependencies between these features may be lost.

4.2.1 Nominal features

kNN cannot handle nominal features well without the use of special metrics. Nominal features can use a special metric which returns 0 if two values are the same, and 1 otherwise. Another approach is to split the nominal feature into dummy features and use one-hot encoding.

For ordinal features, the use of regular distance metrics may make sense, but is typically problematic.

4.2.2 The curse of dimensionality

A high number of features causes problems with many models. **Dimensionality reduction** and **feature selection** are important topics in data science.

For kNN, there are two problems with datasets of a high dimension:

- As the number of features grows, the data becomes sparse and individual samples are far apart. To maintain a certain density as features are added, the size of the dataset has to grow by orders of magnitude, which is usually not possible.
- For commonly used distance metrics, the differences between points which are far apart and points which are closer are weakened as the dimension increases.

4.3 Cross-validation

As mentioned before, a good approach to hyperparameter tuning involves splitting the entry dataset into training, testing, and validation sets. Cross-validation is an alternative approach to this problem.

1. Data is randomly split into training and testing sets

2. ***k*-fold Cross-Validation** – choose $k \geq 2$ and $k \leq |D|$

3. The training set is (randomly) split into k subsets of *roughly* the same size, $D_1, \ldots, D_k$

4. The model (decision tree, kNN, etc.) is trained for given hyperparameter values on the dataset

$$\left(\bigcup_{i=1}^{k} D_i\right) \setminus D_j$$

   where $j = 1, 2, \ldots, k$, and the error e_j (MSE, classification inaccuracy, etc.) is measured on the set D_j.

5. The cross-validation error is computed as

$$\frac{1}{k}\sum_{i=1}^{k} e_i$$

6. The above process is repeated for all desired hyperparameter configurations. The configuration with the lowest cross-validation error wins.

Cross-validation can be tremendously computationally expensive and is therefore not a straightforward replacement of a simple validation set. The extreme case when $k = |D| - 1$ is known as **leave-one-out cross-validation**.

5 Conditional probability in classification and naive Bayes

Let us consider a classification task with p discrete features and a discrete target variable Y. Let $\mathbb{Y}$ be the range of Y and $\boldsymbol{X} = (X_1, \ldots, X_p)^T$ the vector of features with values in χ. Based on the training set, we find an estimate of $P(Y = y|\boldsymbol{X} = x)$ for each $x \in \chi$ and $y \in \mathbb{Y}$. These probabilities can be used to predict Y given a sample x like so:

$$\hat{Y} = \arg\max_{y \in \mathbb{Y}} P(Y = y|\boldsymbol{X} = x),$$

i.e. the most probable value y given some x. This prediction is called a **MAP estimate** (*maximum a posteriori probability*). The question is then how to estimate $P(Y = y|\boldsymbol{X} = x)$. Note that this probability can be rewritten according to the Bayes' theorem from probability theory as

$$P(Y = y|\boldsymbol{X} = x) = \frac{P(\boldsymbol{X} = x|Y = y)P(Y = y)}{P(\boldsymbol{X} = x)}$$

where

$$P(\boldsymbol{X} = x) = \sum_{y \in \mathbb{Y}} P(\boldsymbol{X} = x|Y = y)P(Y = y).$$

This also requires an estimate of $P(Y = y)$, which is, however, arrived at trivially from the training dataset. Since only the value of arg max is of interest to us, we can scale the values passed to this function by any constant and still arrive at the same result. Notably, we can multiply all values by $P(\boldsymbol{X} = x)$, because this probability is constant for all y. This gets rid of the fraction from the application of Bayes' theorem. We express the fact that A is an α-multiple of B for some α independent on y as

$$A \propto B.$$

In this case,

$$P(Y = y|\boldsymbol{X} = x) \propto P(\boldsymbol{X} = x|Y = y)P(Y = y).$$

Therefore, the predicted value $\hat{Y}$ is defined as follows

$$\hat{Y} = \arg\max_{y \in \mathbb{Y}} P(\boldsymbol{X} = x|Y = y)P(Y = y).$$

The last remaining problem is estimating $P(\boldsymbol{X} = x|Y = y)$.

5.1 Naive Bayes classification

The naive Bayes method introduces the following assumption:
Given the condition $Y = y$, all features are independent.
Therefore, $\forall y \in \mathbb{Y}, x = (x_1, \dots, x_p)^T \in \chi$

$$P(\boldsymbol{X} = x|Y = y) = P(X_1 = x_1|Y = y) * \dots * P(X_p = x_p|Y = y).$$

The naive Bayes MAP estimate is given by

$$\hat{Y} = \arg\max_{y \in \mathbb{Y}} \prod_{i=1}^{p} P(X_i = x_i|Y = y)P(Y = y).$$

The naivety of naive Bayes lies in the independence assumption, which is often an oversimplification (if it is not outright wrong). Despite this fact, the method yields surprisingly good results in many cases.

Furthermore, naive Bayes has several desirable properties. The decomposition of $P(\boldsymbol{X} = x|Y = y)$ into a product of marginal probabilities separates the features. The estimate of conditional probability of each feature is thus computed independently of others. This significantly improves resilience to the curse of dimensionality – the estimate of $P(X_i = x_i|Y = y)$ can be arrived at using a fairly small amount of data. The amount of samples required to find these estimates does not grow with the number of features.

Additionally, the independence assumption leads to a poor estimate of $P(\boldsymbol{X} = x|Y = y)$. However, the goal of naive Bayes is not this specific estimate, but the MAP estimate derived from it. The MAP estimate is closer to being correct as long as the real value of y has a higher probability than other values. In practice, this is often the case with the naive Bayes method.

5.2 Naive Bayes models

Let us now consider the problem of estimating $P(\boldsymbol{X} = x|Y = y)$, where $\boldsymbol{X}$ is one of the features. The simplest case is for $\boldsymbol{X}$ being binary, then a suitable model of the conditional distribution $\boldsymbol{X}|Y$ is *Bernoulli's distribution* with the parameter p_y, i.e. $P(\boldsymbol{X} = 1|Y = y) = p_y$, noted $(\boldsymbol{X}|Y = y) \sim Be(p_y)$.

A common formula for estimating p_y is

$$\hat{p}_y = \frac{N_{1,y}}{N_{1,y} + N_{0,y}},$$

where $N_{1,y}$ denotes the amount of (training) samples for $\boldsymbol{X} = 1$ and $Y = y$ (analogically for $N_{0,y}$). In terms of mathematical statistics, this is a **MLE – Maximum Likelihood Estimation** of the parameter of Bernoulli's distribution.

The downside of this estimate is that for very high (or very low) values of p_y, the training set may not contain both values of X_i for $Y = y$. Consequently, $N_{1,y} = 0$ or $N_{0,y} = 0$. In this case, $\hat{p}_y = 0$ (or 1). If this happens, the estimate of conditional probability for x_i of the missing value trivially equal to zero. MAP estimate $\hat{Y}$ can then, regardless of the values of other features, never predict the appropriate value y. This problem can be mitigated by using a different initial distribution.

5.2.1 Intermission – a Bayesian approach to estimates

A classical statistical approach is frequentist – the estimate $\hat{p}$ is based solely on the training set. This approach does not account for (our) expert knowledge of the situation. Bayesian statistics introduces a notion of a *prior distribution* of the estimated parameter, derived from the expert knowledge.

In the case of a binary feature $\boldsymbol{X}$, assume that the parameter p has a prior continuous distribution on $(0, 1)$ specified by the density $f_p(p)$. Its shape defines our initial expectation.

Based on the observed data $x = (x_1, \ldots, x_n)^T$, the so-called *posterior distribution* is determined using the Bayes' theorem:

$$f_p(p|x) = \frac{P(\boldsymbol{X} = x|p) f_p(p)}{P(\boldsymbol{X} = x)}$$

where $P(\boldsymbol{X} = x|p)$ is the probability of observing $\boldsymbol{X} = x$ if p is the correct parameter and

$$P(\boldsymbol{X} = x) = \int_p P(\boldsymbol{X} = x|p) f_p(p) \mathrm{d}p$$

is the centered probability of observing $\boldsymbol{X} = x$. The posterior distribution $f_p(p|x)$ expresses the change in our interpretation based on the observed data. An estimate of a specific value of the parameter p is simply the expected value of the posterior distribution,

$$\hat{p} = \int_p p f_p(p|x) \mathrm{d}p.$$

To utilize a Bayesian approach in the case of Bernoulli's distribution $\boldsymbol{X}|Y = y$, one has to find a suitable prior distribution. This is usually a Beta distribution with parameters α, β. Its density is

$$f_p(p) \propto p^{\alpha-1}(1 - p)^{\beta-1}.$$

The choice $\alpha = \beta = 1$ corresponds with a uniform distribution for which the final estimate using a posterior distribution is

$$\hat{p}_y = \frac{N_{1,y} + 1}{N_{1,y} + N_{0,y} + 2}.$$

This estimate is called *add-one smoothing* (also known as *Laplace's rule of succession*). It clearly doesn't reach $\hat{p}_y = 0$.

5.2.2 Categorical features

To generalize the case of binary features, let us now consider a feature X with k possible values $c_1, \ldots, c_k$. An appropriate model of the conditional distribution $X|Y = y$ is *multinoulli distribution*, denoted $\mathrm{Cat}(p_y)$, where $p_y = (p_{1,y}, \ldots, p_{k,y})^T$ and $p_{1,y}, \ldots, p_{k,y}$ are conditional probabilities of values $c_1, \ldots, c_k$, i.e. $P(X = c_j|Y = y) = p_{j,y}$. The estimate of a k-dimensional parameter p_y is usually

$$\hat{p}_y = (\hat{p}_{1,y}, \ldots, \hat{p}_{k,y})^T \text{ and } \hat{p}_{j,y} = \frac{N_{j,y}}{N_{1,y} + \ldots + N_{k,y}},$$

where $N_{j,y}$ denotes the number of samples for $X = c_j$ and $Y = y$. This is also a **maximum likelihood estimate**, again subject to the problem of $\hat{p}_y \in \{0, 1\}$. A more robust estimate follows from the Bayesian approach:

$$\hat{p}_{j,y} = \frac{N_{j,y} + 1}{N_{1,y} + \ldots + N_{k,y} + k}.$$

5.2.3 Continuous features

Let us now consider a *continuous* feature X. The previously showcased probabilistic approach does not work here, as $P(X = x|Y = y) = 0$ for any x. However, instead of conditional probability, conditional density $f_{X|y}(x)$ can be used. It denotes the density of the variable X conditioned by the event $Y = y$. In other words, it is the density of the distribution function

$$F_{X|y}(x) = P(X \leq x|Y = y).$$

The MAP estimate can be computed as

$$\hat{Y} = \arg\max_{y \in \mathbb{Y}} \prod_{i=1}^{l} P(X_i = x_i|Y = y) \prod_{i=l+1}^{p} f_{X_i|y}(x_i) P(Y = y_i),$$

where $X_1, \ldots, X_l$ are discrete features and $X_{l+1}, \ldots, X_p$ are continuous features.

5.2.4 Gauss' distribution

A common model of the conditional distribution $X|Y = y$ is (for continuous variables) normal distribution $\mathrm{N}(\mu_y, \sigma_y^2)$ with expected value μ_y and variance σ_y^2. The conditional density is

$$f_{X|y}(x) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \mathrm{e}^{-\frac{1}{2\sigma_y^2}(x-\mu_y)^2}.$$

Maximum likelihood estimates for this model are

$$\hat{\mu}_y = \frac{1}{N_y}\sum_{i}^{N_y} x_i \text{ and } \hat{\sigma}_y^2 = \frac{1}{N_y}\sum_{i}^{N_y} (x_i - \hat{\mu}_y)^2 ,$$

where $x_1, \dots, x_{N_y}$ are values of the feature X, for which $Y = y$.

5.2.5 Classifying text with a Bayesian classifier

One of the real-world use-cases of Bayesian classifiers is text classification based on the *bag-of-words* model. An input document is represented as a set of frequencies of individual words of some dictionary $\mathbb{D}$. Each document thus has $D \equiv |\mathbb{D}|$ features $X_1, \dots, X_D$ where X_j denotes the number of occurrences of the j-th word from $\mathbb{D}$. The simplest model for conditional probability is then the following:

$$P(\boldsymbol{X} = x|Y = y) = \frac{n!}{\prod_{j=1}^{D} x_j!}\prod_{j=1}^{D} p_{j,y}^{x_j},$$

where $p_{j,y}$ is the probability that a randomly picked word from a document of the y class will be the j-th word from the dictionary $\mathbb{D}$, and $n = \sum_j x_j$ is the number of words in the document.

For a constant n, the above is called a *multinomial distribution* with parameters n and $p_{1,y}, \dots, p_{D,y}$, where $\sum_j p_{j,y} = 1$.

The remaining task is to estimate the parameters $p_{1,y}, \dots, p_{D,y}$ based on the training dataset of N classified documents. The simplest estimate is the ratio of the frequency of the j-th word in the entire category to the total length of all documents in the category

$$\hat{p}_{j,y} = \frac{N_{j,y}}{N_y},$$

where $N_{j,y} = \sum_{i=1}^{N} x_{i,j}$ and $x_{i,j}$ is the frequency of the j-th word in the i-th document, and finally $N_y = \sum_{j=1}^{D} N_{j,y}$ is the total amount of words in the category y. It is possible to apply a Bayesian approach to this problem as well – the *add-one smoothing* estimate is

$$\hat{p}_{j,y} = \frac{N_{j,y} + 1}{N_y + D}.$$

5.2.6 Remarks

The process of building a model of the joint probability $P(\boldsymbol{X} = x|Y = y)$ is called the *generative approach.* The resulting model is, accordingly, a *generative model.* The term *generative* means that the model of the probability function exposes all information about the distribution from which it was built. A generative model can be used for generation[3] of new observations. A maximum a priori probability estimate based on $P(\boldsymbol{X} = x|Y = y)$ is called a *generative classifier.*

An alternative approach is to estimate $P(Y = y|\boldsymbol{X} = x)$ directly. This approach is called *discriminative* and the associated conditional probability model

[3]Possible overuse of the word “generation” and its siblings.

is thus a *discriminative model.* Analogically, the term *discriminative classifier* refers to a MAP estimate based on $P(Y = y|\boldsymbol{X} = x)$.

6 Linear regression

The linear regression model assumes a linear dependency between the features $X_1, \ldots, X_p$ and the target variable Y. We do not assume that this dependency is perfect (i.e. we account for the fact that some features Y depends on are missing from the dataset), therefore we model it like so:

$$Y = w_1 x_1 + \ldots + w_p x_p + \varepsilon,$$

where $w_1, \ldots, w_p$ are some unknown coefficients and ε is a random variable.

Linear regression model The value of the target variable Y in $(x_1, \ldots, x_p)^T$ is

$$Y = w_0 + w_1 x_1 + \ldots + w_p x_p + \varepsilon,$$

where $\mathrm{E}\varepsilon = 0$. The w_0 coefficient is called the *intercept* and denotes the default value of Y when $\forall i \in (0, p] : x_i = 0$.

By adding a new *constant* feature $X_0 = x_0 = 1$ and adopting vector notation

$$\boldsymbol{x} = (x_0, x_1, \ldots, x_p)^T \text{ and } \boldsymbol{w} = (w_0, w_1, \ldots, w_p)^T,$$

the above definition can be shortened to

$$Y = \boldsymbol{w}^T \boldsymbol{x} + \varepsilon.$$

$\boldsymbol{w}$ is often called a *vector of weights.*

6.1 Predictions using the linear regression model

Assume the existence of an estimate $\hat{\boldsymbol{x}}$ of the vector of weights $\boldsymbol{x}$. The prediction of the value of Y at $\boldsymbol{x}$ is

$$\hat{Y} = \hat{\boldsymbol{w}}^T \boldsymbol{x} = \hat{w}_0 + \hat{w}_1 x_1 + \ldots + \hat{w}_p x_p.$$

The actual value of Y at $\boldsymbol{x}$ is given by

$$Y = \boldsymbol{w}^T \boldsymbol{x} + \varepsilon$$

and is thus a random variable. The precondition $\mathrm{E}\varepsilon = 0$ implies

$$\mathrm{E}Y = \boldsymbol{w}^T \boldsymbol{x}.$$

6.1.1 Prediction error

Let us consider the problem of estimating suitable parameters of the model (the vector $\boldsymbol{w}$). The goal is to find a value for $\boldsymbol{w}$ which minimizes the prediction error. Such a value can then be used as the estimate $\hat{\boldsymbol{w}}$. This requires a formalization of **prediction error** and of its minimization. The prediction error is usually measured using a non-negative function $L : \mathbb{R}^2 \mapsto \mathbb{R}_0^+$, which we call the **loss**

function. We apply L to the real value of variable Y and to the associated prediction $\hat{Y}$. A common choice for L is the *quadratic loss function*,

$$L(Y, \hat{Y}) = (Y - \hat{Y})^2.$$

With the definition of a loss function, we can move on to the actual process of evaluating prediction error of the model at hand. To evaluate the loss function, a sample from the training set is required (otherwise we would not know the real value of Y). To make the most out of the provided training data, let us minimize the sum of errors over the entire training set. For the quadratic loss function, this sum is known as the **residual sum of squares**:

$$\mathrm{RSS}(\boldsymbol{x}) = \sum_{i=1}^{N} L(Y_i, \boldsymbol{w}^T \boldsymbol{x}_i) = \sum_{i=1}^{N} \left(Y_i - \boldsymbol{w}^T \boldsymbol{x}_i\right)^2 .$$

The minimization of this formula gives the estimate $\hat{\boldsymbol{w}}$. This approach is called the **method of least squares**.

6.2 Extremes of multivariate functions

Since $\boldsymbol{w}$ is a vector, minimization of $\mathrm{RSS}(\boldsymbol{w})$ requires some applications of multivariate calculus.

Partial derivative of a function $f : \mathbb{R}^d \to \mathbb{R}$, $f(x_1, \ldots, x_d)$ with respect to the variable x_i at point $\boldsymbol{a} = (a_1, \ldots, a_d) \in \mathbb{R}^d$ is the derivative of the function $g(x_i) = f(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_d)$ at the point a_i denoted by

$$\partial_{x_i} f(\boldsymbol{a}) \text{ or } \frac{\partial f}{\partial x_i}(\boldsymbol{a}) .$$

$\partial_{x_i} f$ (or equivalently $\frac{\partial f}{\partial x_i}$) denotes a function which at every point in which it is finite returns the value of the partial derivative with respect to x_i at this point.

Gradient of a function $f : \mathbb{R}^d \to \mathbb{R}$ (of d variables), where all partial derivatives of f are finite at the point $\boldsymbol{a} \in \mathbb{R}^d$, is the vector

$$\nabla f(\boldsymbol{a}) = \left(\frac{\partial f}{\partial x_1}(\boldsymbol{a}), \ldots, \frac{\partial f}{\partial x_d}(\boldsymbol{a}) \right) .$$

∇f denotes a mapping which for each point, when possible, assigns a gradient to this point.

Hessian matrix Let $f : \mathbb{R}^d \to \mathbb{R}$ be a function of d variables. Then the *Hessian matrix* of the function f at point $\boldsymbol{a} \in \mathbb{R}^d$ is

$$\mathbf{H}_f(\boldsymbol{a}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2}(\boldsymbol{a}) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d}(\boldsymbol{a}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1}(\boldsymbol{a}) & \cdots & \frac{\partial^2 f}{\partial x_d^2}(\boldsymbol{a}) \end{pmatrix} ,$$

where $\frac{\partial^2 f}{\partial x_i \partial x_j}(\boldsymbol{a}) = \left(\frac{\partial}{\partial x_i} (\frac{\partial f}{\partial x_j}) \right) (\boldsymbol{a})$ denotes the second partial derivative with respect to x_i and x_j.

Sufficient condition for the existence of a local extreme Let $f : \mathbb{R}^d \to \mathbb{R}$ be a function of d variables and let $\boldsymbol{x}^* \in \mathbb{R}^d$ such that $\nabla f(\boldsymbol{x}^*) = 0$. If

$$\forall s \in \mathbb{R}^d, s \neq \theta : s^T \mathbf{H}_f(\boldsymbol{x}^*) s > 0$$

where θ is the zero vector, then the function f has a local minimum in $\boldsymbol{x}^*$.

Before employing the above approach for the minimization of RSS, the linear regression model can be rewritten with matrices. The training data is a random selection of samples from the linear model at various points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$. This gives N pairs $(Y_i, \boldsymbol{x}_i)$ where $Y_i = \boldsymbol{w}^T \boldsymbol{x}_i + \varepsilon_i$. Let us now introduce the random vectors $\boldsymbol{Y} = (Y_1, \ldots, Y_N)^T$, $\boldsymbol{\varepsilon} = (\varepsilon_1, \ldots, \varepsilon_n)^T$. Points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ become the rows of the matrix $\boldsymbol{X} \in \mathbb{R}^{N,p+1}$,

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_N^T \end{pmatrix} = \begin{pmatrix} 1 & x_{1;1} & x_{1;2} & \cdots & x_{1;p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N;1} & x_{N;2} & \cdots & x_{N;p} \end{pmatrix}.$$

Using this notation, we can express the entire model of the training set as

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{w} + \boldsymbol{\varepsilon},$$

where $\mathrm{E}\boldsymbol{\varepsilon} = (\mathrm{E}\varepsilon_1, \ldots, \mathrm{E}\varepsilon_N)^T = \theta$.