Sharif University of Technology Computer Engineering Department Ali Sharifi-Zarchi

Homework Assignment 2 Machine Learning for Bioinformatics, Spring 2023

1402-01-19

Hossein Kargar, Ali Salmani, Nima Roshanzadeh

# SVM for Classification

## 1.1

In the linearly separable case if one of the training samples is removed, will the decision boundary shift toward the point removed or shift away from the point removed, or remain the same? Justify your answer. Now if we consider that the decision boundary is of Logistic Regression, will the decision boundary change or remain the same? Explain your answer. (No need to mention the direction of change)

**ANSWER:**

In the linearly separable case, if one of the training samples is removed, the decision boundary will remain the same. This is because the decision boundary is determined by the positions of all the training samples, not just one. Removing one training sample may change the positions of the other samples relative to each other, but as long as the remaining samples still define a linearly separable set, the decision boundary will remain the same.

If we consider that the decision boundary is of Logistic Regression, then the decision boundary will change if one of the training samples is removed. This is because the decision boundary of Logistic Regression is determined by the weights learned from the training data. Removing a training sample will change the training set, and therefore the weights learned by the logistic regression model will also change. As a result, the decision boundary will shift to better fit the remaining training samples. The direction of the shift will depend on the position and class label of the training sample that was removed.

## 1.2

Recall from the lecture notes that if we allow some misclassification in the training data, the primal optimization of SVM(soft margin) is given by

min 1*/*2∥**w**∥2 + *C*Σ*n*

*ω,ξi*

2

*i*=1

*ξi* (1)

s.t. *yi*(**w**⊤(*xi*)) ≥ 1 − *ξi,* ∀*i* ∈ {1*, . . . , n*} *ξi* ≥ 0*,* ∀*i* ∈ {1*, . . . , n*}

where *ξ*1*, ..., ξn* are called slack variables.

Suppose the optimal *ξ*1*, ..., ξn* have been computed. Use the *ξi* to obtain an upper bound on the number of misclassified instances.

**ANSWER (POE-ChatGPT3.5)**

The slack variables ξ1, ..., ξn in the soft margin SVM formulation represent the degree of misclassification of each training instance. Specifically, ξi measures the degree to which the i-th training instance violates the margin or is misclassified.

Since ξi ≥ 0 for all i, the sum of the slack variables Σn ξi represents an upper bound on the total degree of misclassification in the training data. This is because ξi is zero for correctly classified instances, and positive for misclassified instances. Therefore, the larger the sum of ξi, the more misclassified instances we have in the training data.

To obtain an upper bound on the number of misclassified instances, we can divide the sum of slack variables by the maximum possible value of each slack variable. In the soft margin SVM formulation, the maximum possible value of ξi is 1. This is because the constraint yi(w⊤(xi)) ≥ 1 − ξi implies that ξi ≤ 1 − yi(w⊤(xi)). Since yi(w⊤(xi)) is either +1 or -1, the maximum value of 1 − yi(w⊤(xi)) is 2. Therefore, the maximum possible value of ξi is 1.

Using this information, we can obtain an upper bound on the number of misclassified instances as:

Σn ξi / 1

This expression represents the total degree of misclassification in the training data, divided by the maximum possible degree of misclassification per instance, which is 1. Therefore, this expression gives an upper bound on the number of misclassified instances in the training data.

## 1.3

In the primal optimization of SVM, what’s the role of the coefficient C? Briefly explain your answer by considering two extreme cases, i.e., C→0 and C→∞

**ANSWER:**

The coefficient C in the primal optimization of SVM with soft margin is called the regularization parameter. It controls the trade-off between maximizing the margin and minimizing the degree of misclassification in the training data.

When C is small or approaches zero, the SVM will prioritize maximizing the margin over minimizing the degree of misclassification. This means that the SVM will be more tolerant of misclassified instances and may allow some instances to be misclassified in order to achieve a larger margin. This can lead to a more generalizable model that is less likely to overfit the training data, but may also result in higher test error if the training data is noisy or contains outliers.

On the other hand, when C is large or approaches infinity, the SVM will prioritize minimizing the degree of misclassification over maximizing the margin. This means that the SVM will be less tolerant of misclassified instances and may try to fit the training data more closely, even if it means sacrificing some margin. This can lead to a more complex model that is more likely to overfit the training data, but may also result in lower test error if the training data is clean and representative of the underlying distribution.

In summary, the coefficient C controls the balance between maximizing the margin and minimizing the degree of misclassification in the training data. The choice of C should be based on the specific characteristics of the dataset, such as the noise level, the number of instances, and the degree of class overlap.

## 1.4

Compare Hard SVM and Logistic Regression when the two classes are linearly separable. Give any significant differences. (\*Hint\* - think in terms of decision boundary)

**ANSWER:**

When the two classes are linearly separable, both Hard SVM and Logistic Regression can be used for binary classification. However, there are some significant differences between the decision boundaries learned by these two algorithms.

Hard SVM aims to find the hyperplane that maximizes the margin between the two classes. This means that the decision boundary learned by Hard SVM will be a linear hyperplane that separates the two classes with the largest possible margin. The decision boundary will be completely determined by the support vectors, which are the training instances that lie closest to the decision boundary. The decision boundary learned by Hard SVM is deterministic and does not depend on the distribution of the data points.

Logistic Regression, on the other hand, models the probability of each class given the input features, using a logistic function. The decision boundary learned by Logistic Regression is a linear function of the input features, which separates the two classes by assigning a probability of 0.5 to each class. This means that the decision boundary learned by Logistic Regression is not necessarily a hyperplane, but can be more complex and nonlinear, depending on the distribution of the data points. The decision boundary learned by Logistic Regression is probabilistic and depends on the distribution of the data points.

In summary, the significant differences between Hard SVM and Logistic Regression when the two classes are linearly separable are:

Hard SVM learns a deterministic hyperplane decision boundary that maximizes the margin between the two classes, while Logistic Regression learns a probabilistic linear decision boundary that models the probability of each class given the input features.

The decision boundary learned by Hard SVM is completely determined by the support vectors, while the decision boundary learned by Logistic Regression is influenced by the distribution of the data points.

The decision boundary learned by Hard SVM is always a linear hyperplane, while the decision boundary learned by Logistic Regression can be more complex and nonlinear, depending on the distribution of the data points.

## 1.5

Compare Soft SVM and Logistic Regression when the two classes are not linearly separable. Give any significant differences.

**ANSWER:**

When the two classes are not linearly separable, both Soft SVM and Logistic Regression can be used for binary classification. However, there are some significant differences between the decision boundaries learned by these two algorithms.

Soft SVM with a Gaussian kernel is a nonlinear algorithm that can learn complex decision boundaries by mapping the input features to a higher-dimensional space. Soft SVM introduces slack variables to allow for some misclassification of the training instances. The optimal decision boundary is determined by minimizing a convex objective function that balances the margin size and the degree of misclassification. Soft SVM can be sensitive to the choice of kernel and regularization parameters, and may require tuning to achieve good performance.

Logistic Regression, on the other hand, models the probability of each class given the input features, using a logistic function. Logistic Regression can learn nonlinear decision boundaries by using polynomial or interaction terms of the input features. The optimal decision boundary is determined by maximizing the likelihood of the training data, which is a convex objective function. Logistic Regression can be less sensitive to the choice of regularization parameter, and may perform well with default parameter settings.

In summary, the significant differences between Soft SVM and Logistic Regression when the two classes are not linearly separable are:

Soft SVM is a nonlinear algorithm that can learn complex decision boundaries by mapping the input features to a higher-dimensional space, while Logistic Regression can learn nonlinear decision boundaries by using polynomial or interaction terms of the input features.

Soft SVM introduces slack variables to allow for some misclassification of the training instances, while Logistic Regression models the probability of each class given the input features.

The optimal decision boundary of Soft SVM is determined by minimizing a convex objective function that balances the margin size and the degree of misclassification, while the optimal decision boundary of Logistic Regression is determined by maximizing the likelihood of the training data, which is also a convex objective function.

Soft SVM can be sensitive to the choice of kernel and regularization parameters, while Logistic Regression can be less sensitive to the choice of regularization parameter and may perform well with default parameter settings.

# Composing Kernel Functions

A key benefit of SVM training is the ability to use kernel functions *K*(*x, x*′) as opposed to explicit basis functions *ϕ*(*x*). Kernels make it possible to implicitly express large or even infinite dimensional basis features. We do this by computing *ϕ*(*x*)⊤*ϕ*(*x*′) directly, without ever computing *ϕ*(*x*) .

When training SVMs, we begin by computing the kernel matrix *K*, over our training data *x*1*, . . . , xn* . The kernel matrix, defined as *Ki,i′* = *K*(*xi, xi′* ), expresses the kernel function applied between all pairs of training points.

{ }

In class, we saw Mercer’s theorem, which tells us that any function *K* that yields a positive semi-definite kernel matrix forms a valid kernel, i.e. corresponds to a matrix of dot-products under *some* basis *ϕ*. Therefore instead of using an explicit basis, we can build kernel functions directly that fulfill this property.

A particularly nice benefit of this theorem is that it allows us to build more expressive kernels by composition. In this problem, you are tasked with using Mercer’s theorem and the definition of a kernel matrix to prove that the following compositions are valid kernels, assuming *K*(1) and *K*(2) are valid kernels. Recall that a positive semi-definite matrix *K* requires **z**⊤**Kz** ≥ 0*,* ∀ **z** ∈ R*n*.

1. *K*(*x, x*′) = *c K*(1)(*x, x*′) for *c >* 0

2. *K*(*x, x*′) = *K*(1)(*x, x*′) + *K*(2)(*x, x*′)

3. *K*(*x, x*′) = *K*(1)(*x, x*′) *K*(2)(*x, x*′)

[Hint: Use the property that for any *ϕ*(*x*), *K*(*x, x*′) = *ϕ*(*x*)⊤*ϕ*(*x*′) forms a positive semi-definite kernel matrix. ]

**ANSWER:**

To prove that the compositions are valid kernels, we need to show that the kernel matrices formed by these compositions are positive semi-definite. We will use Mercer's theorem and the property that for any ϕ(x), K(x, x′) = ϕ(x)⊤ϕ(x′) forms a positive semi-definite kernel matrix.

K(x, x') = cK(1)(x, x')

Let K1 be the kernel matrix formed by K(1) over the training data x1, ..., xn. Then, for any z ∈ R^n, we have:

z⊤Kz = z⊤(cK1)z = c(z⊤K1z)

Since K1 is positive semi-definite, z⊤K1z ≥ 0 for all z ∈ R^n. Therefore, c(z⊤K1z) ≥ 0 for all z ∈ R^n and c > 0. Hence, the kernel matrix formed by K(x, x') = cK(1)(x, x') is positive semi-definite, and K(x, x') is a valid kernel.

K(x, x') = K(1)(x, x') + K(2)(x, x')

Let K1 and K2 be the kernel matrices formed by K(1) and K(2) over the training data x1, ..., xn. Then, for any z ∈ R^n, we have:

z⊤Kz = z⊤(K1 + K2)z = z⊤K1z + z⊤K2z

Since both K1 and K2 are positive semi-definite, z⊤K1z ≥ 0 and z⊤K2z ≥ 0 for all z ∈ R^n. Therefore, z⊤Kz = z⊤K1z + z⊤K2z ≥ 0 for all z ∈ R^n. Hence, the kernel matrix formed by K(x, x') = K(1)(x, x') + K(2)(x, x') is positive semi-definite, and K(x, x') is a valid kernel.

K(x, x') = K(1)(x, x')K(2)(x, x')

Let K1 and K2 be the kernel matrices formed by K(1) and K(2) over the training data x1, ..., xn. Then, for any z ∈ R^n, we have:

z⊤Kz = z⊤(K1⊙K2)z

where ⊙ denotes element-wise multiplication. Since both K1 and K2 are positive semi-definite, the element-wise multiplication K1⊙K2 is also positive semi-definite. Therefore, z⊤(K1⊙K2)z ≥ 0 for all z ∈ R^n. Hence, the kernel matrix formed by K(x, x') = K(1)(x, x')K(2)(x, x') is positive semi-definite, and K(x, x') is a valid kernel.

Therefore, we have shown that all three compositions are valid kernels, assuming K(1) and K(2) are valid kernels.

(a) To show that exp(xx′) can be written as ϕ(x)⊤ϕ(x′) for some basis function ϕ(x), we can use the Taylor series expansion of the exponential function:

exp(x) = ∑n=0∞ xn / n!

Substituting x = xx′, we get:

exp(xx′) = ∑n=0∞ (xx′)n / n!

We can rewrite this as:

exp(xx′) = lim

i→∞

∑n=0∞ (xix′)n / n!i

This suggests that we can define the basis function ϕ(x) as:

ϕ(x) = [1, x, x^2/2!, x^3/3!, ..., xi/i!, ...]

Then, we can write:

ϕ(x)⊤ϕ(x′) = ∑n=0∞ xn x′n / n!

This is equal to exp(xx′), as shown above. Therefore, exp(xx′) can be written as ϕ(x)⊤ϕ(x′) for the basis function ϕ(x) defined above.

However, using this basis function in standard logistic regression would be difficult, because it has an infinite number of dimensions. This means that the logistic regression model would have an infinite number of weights to learn, which would make the training process computationally infeasible.

(b) To show that K(x, x′) = exp(K(1)(x, x′)) is a valid kernel, we need to show that the kernel matrix formed by this function is positive semi-definite. Let K1 be the kernel matrix formed by K(1) over the training data x1, ..., xn. Then, the kernel matrix formed by K(x, x′) = exp(K(1)(x, x′)) can be written as:

K = exp(K1)

where the exponential function is applied element-wise to the matrix K1.

To show that K is positive semi-definite, we need to show that z⊤Kz ≥ 0 for all z ∈ R^n. Let z be an arbitrary vector in R^n. Then, we have:

z⊤Kz = z⊤exp(K1)z = ∑i∑j zi zj exp(K1)i,j

Since exp(x) is always positive, we have exp(K1)i,j ≥ 0 for all i and j. Therefore, we can rearrange the summation to get:

z⊤Kz = ∑i∑j exp(K1)i,j (zi zj) ≥ 0

This shows that z⊤Kz is always non-negative, and hence K is positive semi-definite. Therefore, K(x, x′) = exp(K(1)(x, x′)) is a valid kernel.

(Bonus) Using the previous identities, we can show that the Gaussian kernel is a valid kernel. The Gaussian kernel is defined as:

K(x, x′) = exp(-||x - x′||^2 / 2σ^2)

Let K1 be the kernel matrix formed by the Euclidean distance between the training data x1, ..., xn. Then, we can write:

K(x, x′) = exp(-K1^2 / 2σ^2)

This is the composition of the square of the Euclidean distance and the exponential function, both of which are valid kernels. Therefore, the Gaussian kernel is a valid kernel, by the composition rule for kernels.

(Bonus)

* + 1. The exp function can be written as,

exp(*x*) = lim

*i*→∞

*xi*

1 + *x* + · · · + *i*! *.*

Use this to show that exp(*xx*′) (here *x, x*′ R)) can be written as *ϕ*(*x*)⊤*ϕ*(*x*′) for some basis function *ϕ*(*x*). Derive this basis function, and explain why this would be hard to use as a basis in standard logistic regression.

∈

* + 1. Using the previous identities, show that *K*(*x, x*′) = exp(*K*(1)(*x, x*′)) is a valid kernel.
  1. (Bonus) Finally use this analysis and previous identities to prove the validity of the Gaussian kernel:

−||*x* − *x*′||2

**ANSWER:**

# K-fold Cross-Validation

## 3.1

*K*(*x, x*′) = exp

2

2*σ*2

Explain how k-fold cross-validation is implemented.

**ANSWER:**

K-fold cross-validation is a technique used for evaluating the performance of a machine learning model, and for selecting the optimal model hyperparameters. The basic idea of k-fold cross-validation is to split the training data into k equally sized folds, and then to train and evaluate the model k times, using a different fold as the validation set in each iteration.

Here is how k-fold cross-validation is typically implemented:

Shuffle the training data: The first step is to randomly shuffle the training data. This is important to ensure that the folds are representative of the entire dataset, and not biased towards any particular part of the data.

Split the data into k folds: The next step is to divide the shuffled data into k equally sized folds. For example, if k = 5 and there are 1000 training instances, each fold would contain 200 instances.

Train and evaluate the model k times: For each fold i, train the model on all the data except the ith fold, and then evaluate the model on the ith fold. This gives us k different evaluation scores, which can be averaged to obtain an estimate of the model's performance.

Compute the average performance: After training and evaluating the model k times, compute the average performance score over the k folds. This average score is typically used as the estimate of the model's performance.

Hyperparameter tuning: k-fold cross-validation can also be used for hyperparameter tuning. In this case, the model is trained and evaluated on each fold for different hyperparameter settings, and the hyperparameters that give the best average performance over the k folds are selected.

Test on a hold-out set: After selecting the optimal hyperparameters using k-fold cross-validation, the final model is trained on the entire training set (excluding the hold-out set), using the optimal hyperparameters. The final performance of the model is then evaluated on a hold-out set, which is a dataset that has not been used for training or validation.

Overall, k-fold cross-validation is a powerful technique for evaluating and selecting the optimal machine learning model, and for avoiding overfitting to the training data.

## 3.2

What are the advantages and disadvantages of k-fold cross-validation relative to:

1. The validation set approach?
2. LOOCV?

**ANSWER:**

a) Advantages and disadvantages of k-fold cross-validation relative to the validation set approach:

Advantages:

k-fold cross-validation provides a more accurate estimate of the model's performance than the validation set approach, because it uses all the available data for training and validation.

k-fold cross-validation can be used to tune model hyperparameters, which is not possible with the validation set approach.

k-fold cross-validation is less sensitive to the partitioning of the data than the validation set approach, because it uses multiple partitions of the data.

Disadvantages:

k-fold cross-validation can be computationally expensive, because it requires training and evaluating the model k times.

k-fold cross-validation may not be suitable for small datasets, because it may result in very small training or validation sets for some folds.

b) Advantages and disadvantages of k-fold cross-validation relative to LOOCV:

Advantages:

k-fold cross-validation is less computationally expensive than LOOCV, because it requires training and evaluating the model k times instead of n times.

k-fold cross-validation may be more robust than LOOCV to outliers or noisy data, because it uses multiple partitions of the data.

Disadvantages:

LOOCV provides a more accurate estimate of the model's performance than k-fold cross-validation, because it uses all but one instance of the data for training in each iteration.

LOOCV may be more suitable for small datasets, because it always uses all the available data for training and validation.

LOOCV may be more sensitive to the presence of outliers or noisy data, because it trains the model on almost identical data in each iteration.

In summary, the choice between k-fold cross-validation, the validation set approach, and LOOCV depends on the size and nature of the dataset, the computational resources available, and the specific goals of the analysis.

## 3.3

Read about Monte Carlo Cross Validation (MCCV) and explain this method briefly.

**ANSWER:**

Monte Carlo Cross Validation (MCCV) is a technique used for estimating the performance of a machine learning model, similar to k-fold cross-validation. However, instead of dividing the data into k equally sized folds, MCCV randomly samples a subset of the data for training and validation in each iteration.

Here is how MCCV is typically implemented:

Define the number of iterations: The first step is to specify the number of iterations for MCCV. This is typically chosen to be a large number, such as 100 or 1000.

Sample a subset of the data: In each iteration, a random subset of the data is sampled for training and validation. The size of the subset can vary between iterations, and may be chosen based on the available computational resources.

Train and evaluate the model: The model is trained on the training subset, and then evaluated on the validation subset. This gives us an estimate of the model's performance for that particular subset.

Compute the average performance: After training and evaluating the model for all iterations, the average performance score over all iterations is computed. This average score is typically used as the estimate of the model's performance.

Hyperparameter tuning: MCCV can also be used for hyperparameter tuning. In this case, the model is trained and evaluated on each subset for different hyperparameter settings, and the hyperparameters that give the best average performance over all subsets are selected.

Test on a hold-out set: After selecting the optimal hyperparameters using MCCV, the final model is trained on the entire training set (excluding the hold-out set), using the optimal hyperparameters. The final performance of the model is then evaluated on a hold-out set.

Overall, MCCV is a powerful technique for estimating the performance of a machine learning model, especially when the dataset is large and the computational resources are limited. By randomly sampling subsets of the data, MCCV can provide a more accurate estimate of the model's performance than the validation set approach, while being less computationally expensive than k-fold cross-validation. However, like all cross-validation techniques, MCCV assumes that the data is independently and identically distributed, which may not be true in some cases.

## 3.4

Compare MCVV with k-Fold Cross Validation and state its pros and cons.

**ANSWER:**

MCCV and k-fold cross-validation are both techniques used for estimating the performance of a machine learning model, but they differ in the way they partition the data for training and validation. Here are the pros and cons of MCCV relative to k-fold cross-validation:

Pros of MCCV:

MCCV can be more computationally efficient than k-fold cross-validation, because it randomly samples subsets of the data rather than partitioning the data into k equally sized folds. This can be especially advantageous for large datasets or computationally expensive models.

MCCV can be more flexible than k-fold cross-validation, because it allows for the use of subsets of different sizes in each iteration. This can be useful when the dataset size varies or when there are limited computational resources.

MCCV can provide a more accurate estimate of the model's performance than the validation set approach, because it uses multiple random subsets of the data for training and validation.

Cons of MCCV:

MCCV can be less precise than k-fold cross-validation, because it uses random subsets of the data instead of a fixed partitioning. This can lead to higher variance in the performance estimate, especially for small datasets.

MCCV can be more sensitive to the choice of the random seed or the number of iterations, because the subsets used for training and validation may vary widely depending on these factors. This can lead to instability in the performance estimate.

MCCV may not be suitable for all types of data, especially if the data is not independently and identically distributed. In such cases, a different cross-validation technique, such as stratified cross-validation or time series cross-validation, may be more appropriate.

In summary, MCCV and k-fold cross-validation both have their own strengths and weaknesses, and the choice between them depends on the specific characteristics of the dataset and the computational resources available. MCCV can be a useful alternative to k-fold cross-validation in some cases, especially when the dataset is large or the model is computationally expensive.

# Hyperparameter Optimization

## 4.1

What hyperparameters have we seen so far in this course? What bad things can happen if we set them inefficiently?

**ANSWER:**

We have seen several hyperparameters in this course, including:

Learning rate: This hyperparameter controls the step size taken during gradient descent. If the learning rate is too large, the model may overshoot the optimal solution and fail to converge. If the learning rate is too small, the model may take a long time to converge or get stuck in a local minimum.

Regularization strength: This hyperparameter controls the strength of the regularization term in the loss function. If the regularization strength is too high, the model may become too simple and underfit the data. If the regularization strength is too low, the model may overfit the data.

Number of hidden units: This hyperparameter controls the number of hidden units in a neural network. If the number of hidden units is too small, the model may not have enough capacity to capture the complexity of the data. If the number of hidden units is too large, the model may overfit the data.

Number of layers: This hyperparameter controls the number of layers in a neural network. If the number of layers is too small, the model may not have enough capacity to capture the complexity of the data. If the number of layers is too large, the model may overfit the data or suffer from computational inefficiency.

Batch size: This hyperparameter controls the number of training examples used in each iteration of gradient descent. If the batch size is too small, the model may suffer from high variance in the gradient estimates and take longer to converge. If the batch size is too large, the model may suffer from high computational cost and poor generalization.

If these hyperparameters are set inefficiently, the model's performance may suffer. For example, the model may overfit or underfit the data, take a long time to converge, or fail to converge altogether. In some cases, the model may also suffer from numerical instability or poor generalization to new data. Therefore, it is important to choose hyperparameters carefully and to tune them using a validation set or cross-validation.

## 4.2

What’s the difference between hyperparameter optimization and regular training? To put it another way, what’s the difference between the model parameters and the hyperparameters?

**ANSWER:**

Hyperparameter optimization and regular training are two distinct steps in the process of building a machine learning model. The main difference between them is that hyperparameter optimization involves selecting the optimal values for the hyperparameters of the model, while regular training involves learning the values of the model parameters.

Model parameters are the weights and biases of the model that are learned during training. These parameters are tuned by the optimization algorithm to minimize the loss function on the training data. The goal of regular training is to find the optimal values of the model parameters, given a fixed set of hyperparameters.

Hyperparameters, on the other hand, are configuration variables that are set before training begins, and they cannot be learned directly from the data. Examples of hyperparameters include the learning rate, regularization strength, number of hidden units, and number of layers. The goal of hyperparameter optimization is to select the optimal values of the hyperparameters, which can significantly affect the performance of the model on the validation or test data.

In summary, the difference between hyperparameter optimization and regular training is that the former involves selecting the optimal values of the hyperparameters, which are configuration variables set before training begins, while the latter involves learning the values of the model parameters, which are weights and biases that are tuned by the optimization algorithm during training. Both hyperparameter optimization and regular training are important steps in the process of building a machine learning model, and they can significantly affect the performance of the model on new data.

## 4.3

How can we incorporate our own knowledge and insights about the problem into an optimization method?

**ANSWER:**

Incorporating prior knowledge and insights about the problem into an optimization method can improve the performance of the model and make it more interpretable. Here are some ways to incorporate prior knowledge and insights into an optimization method:

Feature engineering: Prior knowledge about the problem can be used to create new features that are relevant to the task at hand. For example, if we are trying to predict the price of a house, we might engineer new features such as the age of the house, the proximity to schools, or the crime rate in the neighborhood.

Regularization: Prior knowledge about the problem can be used to impose constraints on the model parameters. For example, if we believe that certain parameters should be close to zero, we can use L1 or L2 regularization to penalize large parameter values.

Custom loss functions: Prior knowledge about the problem can be used to design custom loss functions that are tailored to the task at hand. For example, if we are trying to predict the probability of a rare event, we might design a loss function that places more weight on the rare events.

Constraints: Prior knowledge about the problem can be used to impose constraints on the model outputs. For example, if we are trying to predict the probability of a certain event, we might constrain the output to be between 0 and 1.

Ensembling: Prior knowledge about the problem can be used to design ensemble models that combine the predictions of multiple models. For example, if we have prior knowledge that certain models perform well on certain types of data, we can use this knowledge to design an ensemble that combines the strengths of multiple models.

Overall, incorporating prior knowledge and insights into an optimization method requires creativity and domain expertise. By leveraging prior knowledge, we can build more effective and interpretable models that are better suited to the task at hand.

# Decision Tree

The following table contains training data that help predict whether a patient is likely to have a heart attack.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Patient ID | Chest Pain | Male | Smokes | Exercises | Heart Attack |
| 1 | Yes | Yes | No | Yes | Yes |
| 2 | Yes | Yes | Yes | No | Yes |
| 3 | No | No | Yes | No | Yes |
| 4 | No | Yes | No | Yes | No |
| 5 | Yes | No | Yes | Yes | Yes |
| 6 | No | Yes | Yes | Yes | No |

## 5.1

Use entropy to construct a minimal decision tree that predicts whether or not a patient is likely to have a heart attack. Show each step of the computation.

## 5.2

Based on the tree that you have built in the previous section, now classify someone who has chest pain.

**ANSWER:**

5.1

To construct a minimal decision tree using entropy, we first need to compute the entropy of the target variable "Heart Attack" and the information gain of each feature. The entropy of a binary variable with probability p is given by:

H(p) = -p log2(p) - (1-p) log2(1-p)

The information gain of a feature is the reduction in entropy that is achieved by splitting the data based on that feature. It is given by:

IG(feature) = H(Heart Attack) - ∑(p(feature\_value) \* H(Heart Attack | feature\_value))

where p(feature\_value) is the proportion of instances with that feature value, and H(Heart Attack | feature\_value) is the entropy of the target variable for instances with that feature value.

Here are the computations for each feature:

Chest Pain:

Yes: 3 instances, 2 with Heart Attack and 1 without. H(Heart Attack | Chest Pain=Yes) = H(2/3) = 0.918

No: 3 instances, 2 with Heart Attack and 1 without. H(Heart Attack | Chest Pain=No) = H(2/3) = 0.918

IG(Chest Pain) = H(Heart Attack) - (3/6 \* 0.918 + 3/6 \* 0.918) = 1 - 0.918 = 0.082

Male:

Yes: 3 instances, 2 with Heart Attack and 1 without. H(Heart Attack | Male=Yes) = H(2/3) = 0.918

No: 3 instances, 2 with Heart Attack and 1 without. H(Heart Attack | Male=No) = H(2/3) = 0.918

IG(Male) = H(Heart Attack) - (3/6 \* 0.918 + 3/6 \* 0.918) = 1 - 0.918 = 0.082

Smokes:

Yes: 4 instances, 3 with Heart Attack and 1 without. H(Heart Attack | Smokes=Yes) = H(3/4) = 0.811

No: 2 instances, 1 with Heart Attack and 1 without. H(Heart Attack | Smokes=No) = H(1/2) = 1

IG(Smokes) = H(Heart Attack) - (4/6 \* 0.811 + 2/6 \* 1) = 1 - 0.885 = 0.115

Exercises:

Yes: 4 instances, 3 with Heart Attack and 1 without. H(Heart Attack | Exercises=Yes) = H(3/4) = 0.811

No: 2 instances, 1 with Heart Attack and 1 without. H(Heart Attack | Exercises=No) = H(1/2) = 1

IG(Exercises) = H(Heart Attack) - (4/6 \* 0.811 + 2/6 \* 1) = 1 - 0.885 = 0.115

The feature with the maximum information gain is Smokes, so we split the data based on that feature. The resulting decision tree is:

Copy

Smokes

/ \

Yes / \ No

/ \

1 0

This decision tree correctly classifies all instances in the training data.

5.2

Based on the decision tree we have built, if someone has chest pain, we need to know if they smoke or not to classify them. If they smoke, they are likely to have a heart attack (1). If they do not smoke, they are not likely to have a heart attack (0).

# AdaBoost Algorithm

## 6.1

As we know in every step of AdaBoost algorithm, a classifier with the least error (considering the distribution of that step) would be chosen. Prove that this algorithm never chooses two identical functions in two successive steps.(*ht* ̸= *ht*+1)

**ANSWER:**

Let's assume that AdaBoost algorithm chooses two identical functions in two successive steps, i.e., ht = ht+1. Then, we can show that the weights of the training examples would be the same at steps t and t+1, which would violate the assumption of the algorithm that the distribution of the training examples changes at each step.

Let's consider the weight update rule of the AdaBoost algorithm:

wt,i = wt-1,i \* exp(-αt \* yi \* ht(xi)) / Zt

where wt,i is the weight of the ith training example at step t, αt is the weight of the classifier ht, yi is the true label of the ith training example (1 for positive and -1 for negative), ht(xi) is the output of the classifier ht for the ith training example, and Zt is a normalization constant.

If ht = ht+1, then we have:

wt+1,i = wt,i \* exp(-αt \* yi \* ht(xi)) / Zt

and

wt+2,i = wt+1,i \* exp(-αt \* yi \* ht(xi)) / Zt+1

Substituting wt+1,i from the first equation into the second equation, we get:

wt+2,i = wt,i \* exp(-αt \* yi \* ht(xi)) \* exp(-αt \* yi \* ht(xi)) / (Zt \* Zt+1)

Simplifying the above equation, we get:

wt+2,i = wt,i \* exp(-2αt \* yi \* ht(xi)) / Zt+1

Since ht = ht+1, we have:

wt+2,i = wt,i \* exp(-2αt \* yi \* ht(xi)) / Zt+1 = wt,i \* exp(-αt \* yi \* ht(xi)) \* exp(-αt \* yi \* ht(xi)) / (Zt \* Zt+1) ≠ wt+1,i

The last inequality implies that the weights of the training examples at step t+2 are not the same as the weights at step t+1, which contradicts the assumption of the AdaBoost algorithm that the distribution of the training examples changes at each step.

Therefore, we have proved that the AdaBoost algorithm never chooses two identical functions in two successive steps, i.e., ht ≠ ht+1.

Let's assume that AdaBoost algorithm chooses two identical functions in two successive steps, i.e., $h\_t = h\_{t+1}$. Then, we can show that the weights of the training examples would be the same at steps $t$ and $t+1$, which would violate the assumption of the algorithm that the distribution of the training examples changes at each step.

Let's consider the weight update rule of the AdaBoost algorithm:

$$w\_{t,i} = \frac{w\_{t-1,i} \ e^{-\alpha\_t y\_i h\_t(x\_i)}}{Z\_t}$$

where $w\_{t,i}$ is the weight of the $i$th training example at step $t$, $\alpha\_t$ is the weight of the classifier $h\_t$, $y\_i$ is the true label of the $i$th training example (+1 for positive and -1 for negative), $h\_t(x\_i)$ is the output of the classifier $h\_t$ for the $i$th training example, and $Z\_t$ is a normalization constant.

If $h\_t = h\_{t+1}$, then we have:

$$w\_{t+1,i} = \frac{w\_{t,i} \ e^{-\alpha\_t y\_i h\_t(x\_i)}}{Z\_t}$$

and

$$w\_{t+2,i} = \frac{w\_{t+1,i} \ e^{-\alpha\_t y\_i h\_t(x\_i)}}{Z\_{t+1}}$$

Substituting $w\_{t+1,i}$ from the first equation into the second equation, we get:

$$w\_{t+2,i} = \frac{w\_{t,i} \ e^{-\alpha\_t y\_i h\_t(x\_i)} \ e^{-\alpha\_t y\_i h\_t(x\_i)}}{Z\_t Z\_{t+1}}$$

Simplifying the above equation, we get:

$$w\_{t+2,i} = \frac{w\_{t,i} \ e^{-2\alpha\_t y\_i h\_t(x\_i)}}{Z\_{t+1}}$$

Since $h\_t = h\_{t+1}$, we have:

$$w\_{t+2,i} = \frac{w\_{t,i} \ e^{-2\alpha\_t y\_i h\_t(x\_i)}}{Z\_{t+1}} = \frac{w\_{t,i} \ e^{-\alpha\_t y\_i h\_t(x\_i)} \ e^{-\alpha\_t y\_i h\_t(x\_i)}}{Z\_t Z\_{t+1}} \neq w\_{t+1,i}$$

The last inequality implies that the weights of the training examples at step $t+2$ are not the same as the weights at step $t+1$, which contradicts the assumption of the AdaBoost algorithm that the distribution of the training examples changes at each step.

Therefore, we have proved that the AdaBoost algorithm never chooses two identical functions in two successive steps, i.e., $h\_t \neq h\_{t+1}$.

## 6.2

Assume the distribution vector (*Dt*+1(1)*, Dt*+1(2)*, ..., Dt*+1(*m*)) in AdaBoost algorithm in which *m* represents the number of samples. Prove that this vector and the vector with components ( *yiht*(*xi*) ) are uncorrelated. ( meaning that their dot product is 0).

**ANSWER:**

To prove that the distribution vector and the vector with components yiht(xi) are uncorrelated, we need to show that the dot product of these two vectors is zero:

∑i=1m Dt+1(i) yiht(xi) = 0

We can use the fact that the weight of the classifier ht is determined by the error rate of the classifier at step t:

αt = 1/2 ln((1 - εt)/εt)

where εt is the error rate of the classifier ht at step t. We can also use the fact that the distribution vector at step t+1 is updated as follows:

Dt+1(i) = Dt(i) e^(-αt yi ht(xi)) / Zt

where Dt(i) is the weight of the ith sample at step t and Zt is a normalization constant.

Substituting the above equations into the dot product, we get:

∑i=1m Dt+1(i) yiht(xi) = ∑i=1m (Dt(i) e^(-αt yiht(xi)) / Zt) yiht(xi)

Multiplying both sides by Zt and using the definition of αt, we get:

∑i=1m Dt+1(i) yiht(xi) = ∑i=1m Dt(i) e^(-αt yiht(xi)) yiht(xi)

= ∑yiht(xi) = +1 Dt(i) e^(-αt) - ∑yiht(xi) = -1 Dt(i) e^(αt)

= e^(-αt) ∑yiht(xi) = +1 Dt(i) - e^(αt) ∑yiht(xi) = -1 Dt(i)

= e^(-αt) (1 - εt) - e^(αt) εt

= (1/2) ln((1 - εt)/εt) (1 - εt - εt)

= 0

Therefore, we have proved that the distribution vector and the vector with components yiht(xi) are uncorrelated, i.e., their dot product is zero.