Contents

1 Training Simple Machine Learning Algorithms for Classification 1

1.1 Implementing a perceptron learning algorithm in Python 1

1.1.1 An object-oriented perceptron API 1

1.2 Adaptive linear neurons and the convergence of learning 1

1.2.1 Improving gradient descent through feature scaling 2

1.2.2 Large-scale machine learning and stochastic gradient descent . 3

2 A Tour of Machine Learning Classifiers Using Scikit-Learn 4

2.1 Modeling class probabilities via logistic regression 4

2.1.1 Logistic regression and conditional probabilities 4

2.1.2 Learning the model weights via the logistic loss function . . . 4

2.1.3 Tackling overfitting via regularization 5

2.2 Maximum margin classification with support vector machines 6

2.2.1 Dealing with a nonlinearly separable case using slack variables 6

2.2.2 Alternative implementations in scikit-learn 7

2.3 Solving nonlinear problems using a kernel SVM 7

2.3.1 Kernel methods for linearly inseparable data 7

2.3.2 Using the kernel trick to find separating hyperplanes in a high-dimensional space 8

2.4 Decision tree learning 9

2.4.1 Maximizing IG – getting the most bang for your buck 9

2.4.2 Building a decision tree 9

2.4.3 10

2.5 K-nearest neighbors – a lazy learning algorithm 10

3 Building Good Training Datasets – Data Preprocessing 12

3.1 Dealing with missing data 12

3.1.1 Imputing missing values 12

3.1.2 Understanding the scikit-learn estimator API 12

3.2 Handling categorical data 12

Chapter 1

Training Simple Machine Learning Algorithms for Classification

1.1 Implementing a perceptron learning algorithm in Python

1.1.1 An object-oriented perceptron API

If all the weights are initialized to zero, the learning rate parameter, η, affects only the scale of the weight vector, not the direction. Consider a vector,$v_1 = [1\ 2\ 3]$, where the angle between v_1 and a vector, $v_2 = 0.5 \times v_1$, would be exactly zero.

1.2 Adaptive linear neurons and the convergence of learning

We will take a look at another type of single-layer **neural network (NN): ADAptive LInear NEuron (Adaline)**. The Adaline algorithm is particularly interesting because it illustrates the key concepts of defining and minimizing continuous loss functions.

The key difference between the Adaline rule (also known as the Widrow-Hoff rule) and Rosenblatt's perceptron is that the weights are updated based on a linear activation function rather than a unit step function like in the perceptron. In Adaline, this linear activation function, $\sigma(z)$, is simply the identity function of the net input, so that $\sigma(z) = z$.

As Figure 1.1 indicates, the Adaline algorithm compares the true class labels with the linear activation function's continuous valued output to compute the model error and update the weights. In contrast, the perceptron compares the true class labels to the predicted class labels.

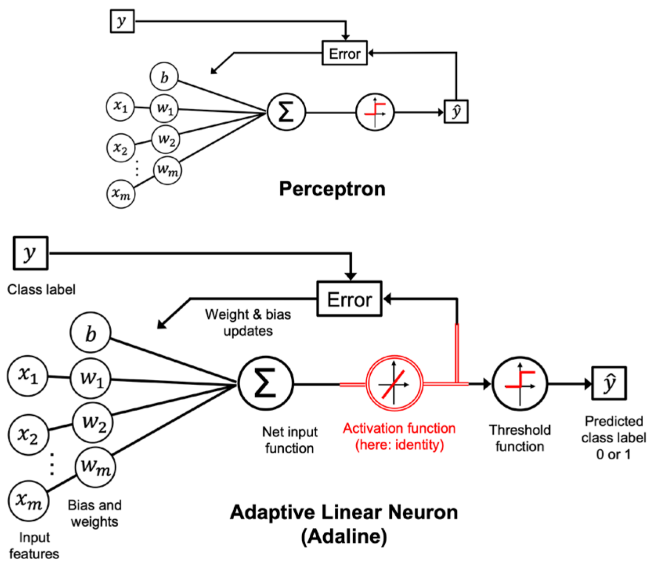

Figure 1.1: A comparison between a perceptron and the Adaline algorithm

Although the Adaline learning rule looks identical to the perceptron rule, we should note that $\sigma(z^{(i)})$ with $\sigma(z^{(i)}) = \mathbf{w}^T\mathbf{x}^{(i)} + b$ is a real number and not an integer class label. Furthermore, the weight update is calculated based on all examples in the training dataset (instead of updating the parameters incrementally after each training example), which is why this approach is also referred to as **batch gradient descent**. To be more explicit and avoid confusion when talking about related concepts later, we will refer to this process as **full batch gradient descent**.

1.2.1 Improving gradient descent through feature scaling

Gradient descent is one of the many algorithms that benefit from feature scaling. In this section, we will use a feature scaling method called standardization. This normalization procedure helps gradient descent learning to converge more quickly; however, it does not make the original dataset normally distributed.

One of the reasons why standardization helps with gradient descent learning is that it is easier to find a learning rate that works well for all weights (and the bias). If the features are on vastly different scales, a learning rate that works well for updating one weight might be too large or too small to update the other weight equally well. Overall, using standardized features can stabilize the training such that the optimizer has to go through fewer steps to find a good or optimal solution (the global loss minimum).

1.2.2 Large-scale machine learning and stochastic gradient descent

A popular alternative to the batch gradient descent algorithm is **stochastic gradient descent (SGD)**, which is sometimes also called iterative or online gradient descent. Instead of updating the weights based on the sum of the accumulated errors over all training examples, $\mathbf{x}^{(i)}$:

$$\Delta w_j = \frac{2\eta}{n} \sum_i \left(y^{(i)} - \sigma(z^{(i)}) \right) x_j^{(i)}$$

we update the parameters incrementally for each training example, for instance:

$$\Delta w_j = \eta \left(y^{(i)} - \sigma(z^{(i)}) \right) x_j^{(i)}, \; \Delta b = \eta \left(y^{(i)} - \sigma(z^{(i)}) \right)$$

Adjusting the learning rate during training

In SGD implementations, the fixed learning rate, η , is often replaced by an adaptive learning rate that decreases over time, for example:

$$\frac{c_1}{[number\ of\ iterations] + c_2}$$

where $c_1$and c_2 are constants. Note that SGD does not reach the global loss minimum but an area very close to it. And using an adaptive learning rate, we can achieve further annealing to the loss minimum.

Chapter 2

A Tour of Machine Learning Classifiers Using Scikit-Learn

2.1 Modeling class probabilities via logistic regression

2.1.1 Logistic regression and conditional probabilities

Under the logistic model, we assume that there is a linear relationship between the weighted inputs and the log-odds:

$$logit(p) = w_1x_1 + w_2x_2 + \cdots + w_mx_m + b = \sum_{i=j} w_jx_j = \mathbf{w}^T\mathbf{x} + b \tag{2.1}$$

2.1.2 Learning the model weights via the logistic loss function

To explain how we can derive the loss function for logistic regression, let's first define the likelihood, $\mathscr{L}$, that we want to maximize when we build a logistic regression model, assuming that the individual examples in our dataset are independent of one another. The formula is as follows:

$$\begin{aligned} \mathscr{L}(\mathbf{w},b|\mathbf{x}) = p(y|\mathbf{x};\mathbf{w},b) &= \prod_{i=1}^{m} p\left(y^{(i)}|\mathbf{x}^{(i)};\mathbf{w},b\right) \\ &= \prod_{i=1}^{m}\left(\sigma(z^{(i)})\right)^{y^{(i)}}\left(1-\sigma(z^{(i)})\right)^{1-y^{(i)}} \end{aligned} \tag{2.2}$$

In practice, it is easier to maximize the (natural) log of this equation, which is called the **log-likelihood** function:

$$\begin{aligned} l(\mathbf{w},b|\mathbf{x}) &= \log \mathscr{L}(\mathbf{w},b|\mathbf{x}) \\ &= \sum_{i=1}\left[y^{(i)}\log(\sigma(z^{(i)})) + (1-y^{(i)})\log(1-\sigma(z^{(i)}))\right] \end{aligned} \tag{2.3}$$

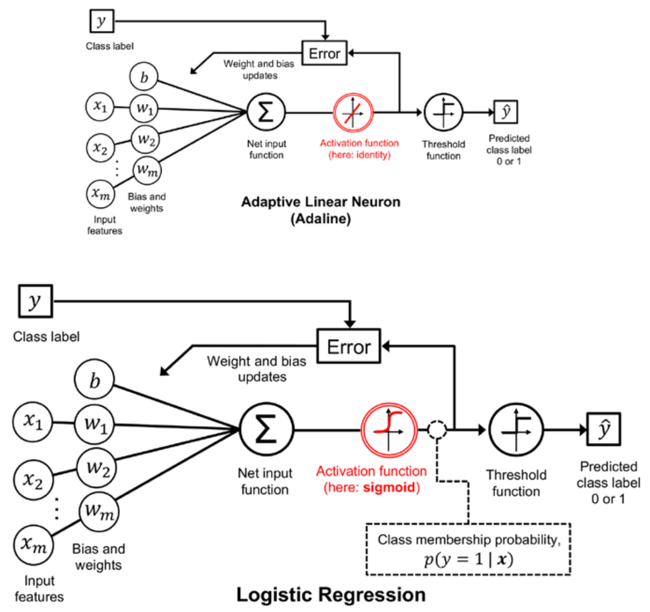
y
Class label
Error
Weight and bias updates
b
x_1
w_1
x_2
w_2
w_m
x_m
Bias and weights
Input features
Net input function
Activation function (here: identity)
Threshold function
Predicted class label 0 or 1
$\hat{y}$
Adaptive Linear Neuron (Adaline)
Activation function (here: sigmoid)
Class membership probability, $p(y = 1 \mid \boldsymbol{x})$
Logistic Regression

Figure 2.1: Logistic regression compared to Adaline

2.1.3 Tackling overfitting via regularization

Overfitting is a common problem in machine learning, where a model performs well on training data but does not generalize well to unseen data (test data). If a model suffers from overfitting, we also say that the model has a high variance, which can be caused by having too many parameters, leading to a model that is too complex given the underlying data. Similarly, our model can also suffer from underfitting (high bias), which means that our model is not complex enough to capture the pattern in the training data well and therefore also suffers from low performance on unseen data.

One way of finding a good bias-variance tradeoff is to tune the complexity of the model via regularization. Regularization is a very useful method for handling collinearity (high correlation among features), filtering out noise from data, and eventually preventing overfitting.

Regularization and feature normalization

Regularization is another reason why feature scaling such as standardization is important. For regularization to work properly, we need to ensure that all our features are on comparable scales.

The loss function for logistic regression can be regularized by adding a simple regularization term, which will shrink the weights during model training:

$$L(\mathbf{w}, b|\mathbf{x}) = \sum_{i=1} \left[y^{(i)} \log(\sigma(z^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(z^{(i)})) \right] + \frac{\lambda}{2n} ||\mathbf{w}||^2 \qquad (2.4)$$

The parameter, C, that is implemented for the LogisticRegression class in scikit-learn comes from a convention in support vector machines. The term C is inversely proportional to the regularization parameter, λ . Consequently, decreasing the value of the inverse regularization parameter, C, means that we are increasing the regularization strength.

The bias-variance tradeoff

Often, researchers use the terms "bias" and "variance" or "bias-variance trade-off" to describe the performance of a model—that is, you may stumble upon talks, books, or articles where speople say that a model has a "high variance" or "high bias." So, what does that mean? In general, we might say that "high variance" is proportional to overfitting and "high bias" is proportional to underfitting.
In the context of machine learning models, variance measures the consistency (or variability) of the model prediction for classifying a particular example if we retrain the model multiple times, for example, on different subsets of the training dataset. We can say that the model is sensitive to the randomness in the training data. In contrast, bias measures how far off the predictions are from the correct values in general if we rebuild the model multiple times on different training datasets; bias is the measure of the systematic error that is not due to randomness.

2.2 Maximum margin classification with support vector machines

In SVMs, our optimization objective is to maximize the margin. The margin is defined as the distance between the separating hyperplane (decision boundary) and the training examples that are closest to this hyperplane, which are the so-called support vectors.

2.2.1 Dealing with a nonlinearly separable case using slack variables

The motivation for introducing the slack variable was that the linear constraints in the SVM optimization objective need to be relaxed for nonlinearly separable data to allow the convergence of the optimization in the presence of misclassifications, under appropriate loss penalization.

The use of the slack variable, in turn, introduces the variable, which is commonly referred to as C in SVM contexts. We can consider C as a hyperparameter for controlling the penalty for misclassification. Large values of C correspond to large error

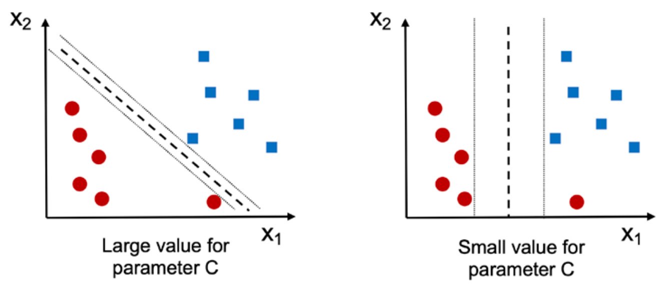

Figure 2.2: The impact of large and small values of the inverse regularization strength C on classification

penalties, whereas we are less strict about misclassification errors if we choose smaller values for C. We can then use the C parameter to control the width of the margin and therefore tune the bias-variance tradeoff, as illustrated in Figure 2.2:

Logistic regression versus SVMs

In practical classification tasks, linear logistic regression and linear SVMs often yield very similar results. Logistic regression tries to maximize the conditional likelihoods of the training data, which makes it more prone to outliers than SVMs, which mostly care about the points that are closest to the decision boundary (support vectors). On the other hand, logistic regression has the advantage of being a simpler model and can be implemented more easily, and is mathematically easier to explain. Furthermore, **logistic regression models can be easily updated, which is attractive when working with streaming data**.

2.2.2 Alternative implementations in scikit-learn

The advantage of using LIBLINEAR and LIBSVM over, for example, native Python implementations is that they allow the extremely quick training of large amounts of linear classifiers. However, sometimes our datasets are too large to fit into computer memory. Thus, scikit-learn also offers alternative implementations via the SGDClassifier class, which also supports online learning via the `partial_fit` method.

2.3 Solving nonlinear problems using a kernel SVM

2.3.1 Kernel methods for linearly inseparable data

The basic idea behind kernel methods for dealing with such linearly inseparable data is to create nonlinear combinations of the original features to project them onto a higher-dimensional space via a mapping function, ϕ , where the data becomes linearly separable. As shown in Figure 3.14, we can transform a two-dimensional dataset into a new

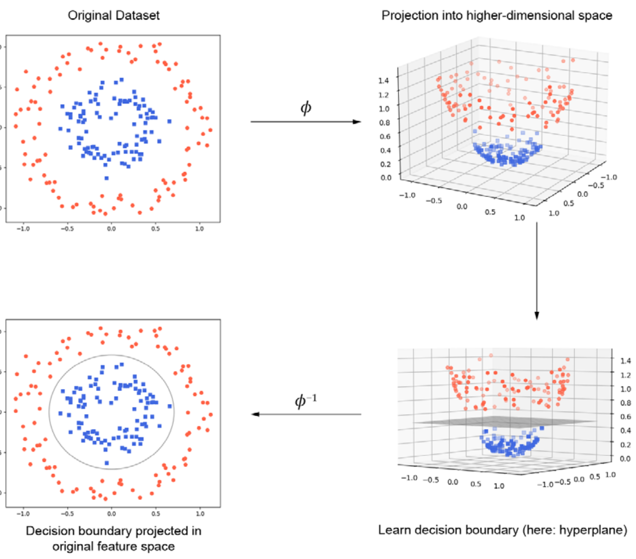
Original Dataset
Projection into higher-dimensional space
ϕ
ϕ^{-1}
Decision boundary projected in original feature space
Learn decision boundary (here: hyperplane)

Figure 2.3: The process of classifying nonlinear data using kernel methods

three-dimensional feature space, where the classes become separable via the following projection:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

2.3.2 Using the kernel trick to find separating hyperplanes in a highdimensional space

To solve a nonlinear problem using an SVM, we would transform the training data into a higher-dimensional feature space via a mapping function, ϕ , and train a linear SVM model to classify the data in this new feature space. Then, we could use the same mapping function, ϕ , to transform new, unseen data to classify it using the linear SVM model.

In practice, we just need to replace the dot product $\mathbf{x}^{(i)T}\mathbf{x}^{(j)}$ by $\phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$. To save the expensive step of calculating this dot product between two points explicitly, we define a so-called kernel function:

$$k(\mathbf{x}^{(i)}\mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$$

Roughly speaking, the term “kernel” can be interpreted as a similarity function between a pair of examples. The minus sign inverts the distance measure into a similarity score, and, due to the exponential term, the resulting similarity score will fall into a range between 1 (for exactly similar examples) and 0 (for very dissimilar examples).

The γ parameter can be understood as a cut-off parameter for the Gaussian sphere. If we increase the value for γ, we increase the influence or reach of the training examples, which leads to a tighter and bumpier decision boundary.

2.4 Decision tree learning

Using the decision algorithm, we start at the tree root and split the data on the feature that results in the largest **information gain (IG)**.

2.4.1 Maximizing IG – getting the most bang for your buck

To split the nodes at the most informative features, we need to define an objective function to optimize via the tree learning algorithm. Here, our objective function is to maximize the IG at each split, which we define as follows:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j) \tag{2.5}$$

Here, f is the feature to perform the split; D_p and D_j are the dataset of the parent and jth child node; I is our **impurity** measure; N_p is the total number of training examples at the parent node; and N_j is the number of examples in the jth child node. As we can see, the information gain is simply the difference between the impurity of the parent node and the sum of the child node impurities—the lower the impurities of the child nodes, the larger the information gain.

The three impurity measures or splitting criteria that are commonly used in binary decision trees are **Gini impurity** (I_G), **entropy** (I_H), and the **classification error** (I_E). Let's start with the definition of entropy for all non-empty classes ($p(i|t) \neq 0$):

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t) \tag{2.6}$$

The Gini impurity can be understood as a criterion to minimize the probability of misclassification:

$$I_G(t) = \sum_{i=1}^{c} p(i|t)(1 - p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2 \tag{2.7}$$

Another impurity measure is the classification error:

$$I_E(t) = 1 - \max p(i|t) \tag{2.8}$$

This is a useful criterion for pruning, but not recommended for growing a decision tree, since it is less sensitive to changes in the class probabilities of the nodes.

2.4.2 Building a decision tree

Although feature scaling may be desired for visualization purposes, note that feature scaling is not a requirement for decision tree algorithms.

2.4.3

The idea behind a random forest is to average multiple (deep) decision trees that individually suffer from high variance to build a more robust model that has a better generalization performance and is less susceptible to overfitting. The random forest algorithm can be summarized in following:

Algorithm 1: The random forest algorithm

```
1  begin
2      for $k \leftarrow 1$ to $K$ do
3          Draw a random bootstrap sample of size $n$ (randomly choose $n$
             examples from the training dataset with replacement);
4          Grow a decision tree from the bootstrap sample; foreach node do
5              Randomly select $d$ features without replacement;
6              Split the node using the feature that provides the best split
                 according to the objective function, for instance, maximizing the
                 information gain;
7          end
8      end
9      Aggregate the prediction by each tree to assign the class label by majority
         vote;
10 end
```

2.5 K-nearest neighbors – a lazy learning algorithm

KNN is a typical example of a lazy learner. It is called "lazy" not because of its apparent simplicity, but because it doesn't learn a discriminative function from the training data but memorizes the training dataset instead.

The KNN algorithm itself is fairly straightforward and can be summarized by the following:

Algorithm 2: The KNN algorithm

```
1  begin
2      Choose the number of $k$ and a distance metric;
3      Find the $k$-nearest neighbors of the data record that we want to classify;
4      Assign the class label by majority vote;
5  end
```

In the case of a tie, the scikit-learn implementation of the KNN algorithm will prefer the neighbors with a closer distance to the data record to be classified. If the neighbors have similar distances, the algorithm will choose the class label that comes first in the training dataset.

Alternative machine learning implementations with GPU support

If you have a computer equipped with an NVIDIA GPU that is compatible with recent versions of NVIDIA's CUDA library, we recommend considering the RAPIDS ecosystem. For instance, RAPIDS' cuML library implements many of scikit-learn's machine learning algorithms with GPU support to accelerate the processing speeds. You can find an introduction to cuML.

Chapter 3

Building Good Training Datasets – Data Preprocessing

3.1 Dealing with missing data

3.1.1 Imputing missing values

One of the most common interpolation techniques is **mean imputation**, where we simply replace the missing value with the mean value of the entire feature column. A convenient way to achieve this is by using the SimpleImputer class from scikit-learn.

3.1.2 Understanding the scikit-learn estimator API

The SimpleImputer class is part of the so-called **transformer** API in scikit-learn, which is used for implementing Python classes related to data transformation. The two essential methods of those estimators are fit and transform. The fit method is used to learn the parameters from the training data, and the transform method uses those parameters to transform the data. Any data array that is to be transformed needs to have the same number of features as the data array that was used to fit the model.

The **classifiers** that we used belong to the so-called estimators in scikit-learn, with an API that is conceptually very similar to the scikit-learn transformer API. Estimators have a predict method but can also have a transform method.

3.2 Handling categorical data

When we are talking about categorical data, we have to further distinguish between ordinal and nominal features. Ordinal features can be understood as categorical values that can be sorted or ordered. For example, t-shirt size would be an ordinal feature, because we can define an order: $XL > L > M$. In contrast, nominal features don't imply any order.

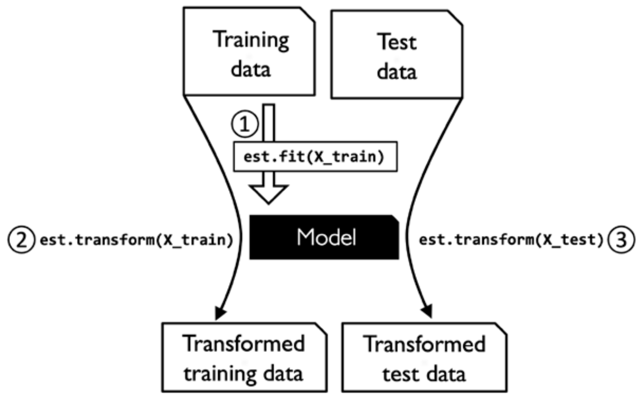

Figure 3.1: Using the scikit-learn API for data transformation

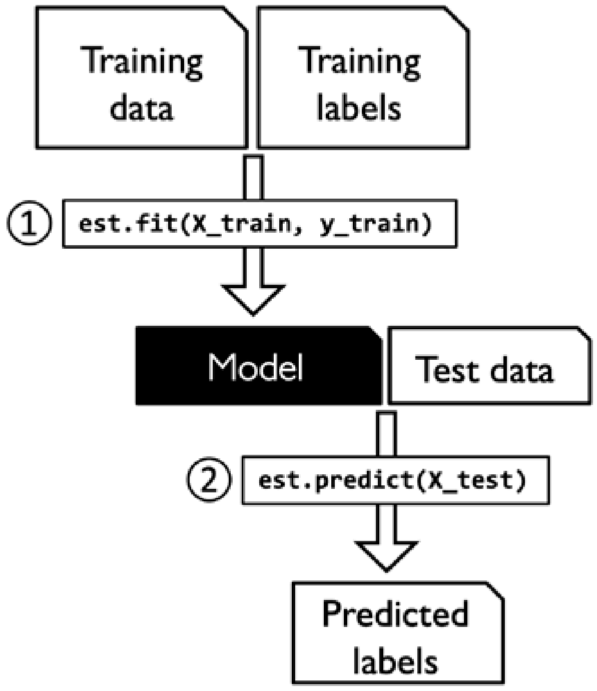

Figure 3.2: Using the scikit-learn API for predictive models such as classifiers

3.2.1 Mapping ordinal features

To make sure that the learning algorithm interprets the ordinal features correctly, we need to convert the categorical string values into integers. Unfortunately, there is no convenient function that can automatically derive the correct order of the labels of our size feature, so we have to define the mapping manually.

Optional: encoding ordinal features