Chapter 19
Artificial Intelligence and Machine Learning

Hamidreza Moradi

Learning Objectives
After completing this chapter, you will be able to:

- Understand and implement machine learning models for regression, classification, and clustering tasks
- Utilize both machine learning and deep learning to devise a predictive model
- Apply concepts of computer vision and natural language processing

1 Introduction

Machine learning (ML) has gained a lot of attention in recent years, and many tools and libraries have been developed to help enthusiasts. Devising a predictive model requires a good understanding of the underlying concepts and hands-on experience with at least a library or package for developing models. In the first section, we aim to discuss traditional ML methods and the libraries to be utilized for developing a prediction model. The models we considered were regression, classification, and clustering. In regression, we aim to predict continuous values, while with classification, the goal is to predict data labels assigned to data instances. And clustering tries to find a natural groping in the data. In the second section, we will start by discussing the basic building blocks of a neural network (NN) model. Then, we develop regression and classification models using NNs to get familiar with how the same task can be achieved using traditional and new approaches. Next, we will review more advanced examples that can be better addressed using NNs, computer vision, and natural language processing (NLP). In computer vision, convolutions will be introduced with examples, and we will see how they can be combined to extract features from an image. In NLP, we will address how we can make words and sentences interpretable and understandable by machines. Finally, the last two sections

H. Moradi (✉)
Department of Computer Science, North Carolina Agricultural and Technical State University, Greensboro, North Carolina, United States
e-mail: hmoradi@ncat.edu

A. K. Mitra (ed.), *Statistical Approaches for Epidemiology*,
https://doi.org/10.1007/978-3-031-41784-9_19

will briefly discuss how we can make ML models interpretable and recent advancements. All code examples are provided here in Python, as it is the preferred language for both ML and deep learning (DL) for students and engineers in academia and industry. Simplicity, flexibility, platform independence, being open-source, and the existence of powerful libraries/frameworks supported by industry leaders are just a few reasons for its popularity and attention.

2 Machine Learning

In artificial intelligence (AI), computer systems are programmed to mimic human behaviors to devise intelligent machines. This required writing programs with the rules needed to make decisions based on the inputs to the system. However, with increased data and computation power, research studies investigate algorithms that, given the inputs and expected outputs, can automatically infer the rules needed, learning from examples. This marked the beginning of the ML era, a subset of AI. In the following subsections, we will discuss some of the ML algorithms that can be used for inferring the rules and making a prediction for the expected output.

2.1 Regression

Regression is one of the early forms of ML models that establishes a linear relationship between a dependent variable and one or more independent variables, usually called features. Linear regression can be graphically presented using a straight line and can be used to predict continuous values. In its simplest form, it can be represented in the form of $y^{'} = \beta + \alpha X$, where α is the weight for the input feature X, β is the intercept for the line, and y' is the predicted value. With ML, the aim is to find α and β using a dataset of samples. Figure 19.1 shows the data samples used to find a regression line with black dots and the devised regression line in blue. The blue dots on the regression line show a couple of predicted values for the corresponding input (X) on the regression line.

The error between the actual observations and their predicted values should be minimized to find the regression line with the best fit. As a result, we are interested in finding the values for α and β that minimize the prediction error $\varepsilon = y_i - y_i^{'}$ for all the observations, where y is the actual observation, $y^{'}$ is the predicted value, and i is the observation number in the sample dataset. We need to define and minimize a cost (loss) function here. The cost function for linear regression can be defined as $\mathcal{L} = \sum \left(y_i - y_i^{'} \right)^2$ to penalize large errors and should be minimized by an ML algorithm.$_i$ In regression, when more than one independent variable exists, the formula can be extended to the form $y^{'} = \beta + \alpha_1 X_1 + \alpha_2 X_2 + \ldots + \alpha_n X_n$, where n is equal to the number of features for each observation. Then, the ML algorithm needs to find the best α_i that minimize the cost function.

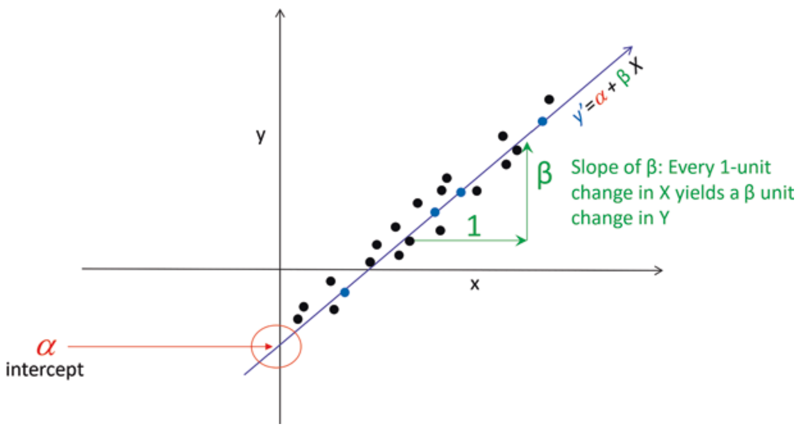
y
x
y'=α+βX
β
1
Slope of β: Every 1-unit change in X yields a β unit change in Y
α
intercept

Fig. 19.1 Regression line with actual and predicted observations

Let's see how to develop an ML model. To train a linear regression model in Python [1], Scikit-Learn [2] library provides many preimplemented functions. These functions can be easily utilized to fit a model to a dataset. After installing the latest versions of Python (v3.8) and Scikit-learn (v1.0.2), we need to import the linear models from the library into the working environment in the code editor of choice.[1] Scikit-Learn also provides many small toy datasets [3] to help users try and learn about the implemented functionalities. Here we will import and use the diabetes dataset. Features include patients' baseline measurements such as body mass index, age, blood, and glucose level to predict a quantitative measure of disease progression a year after the baseline. As the outcomes considered for patients are integer numbers (between 25 and 346), we will use linear regression for modeling and prediction.

```
from sklearn import linear_model
from sklearn import datasets
diabetes = datasets.load_diabetes()
input = diabetes.data
outcome = diabetes.target
```

Train-Test split

In ML, the goal is to train a predictive model and then evaluate the accuracy of an unseen data set. This gives us a measure of how the model may perform in the production environment. To achieve this goal, we can split our data into nonoverlapping train and test subsets of 80% and 20%, respectively.

[1] PyCharm or Jupyter Notebook are recommended.

To have the most accurate model and find the weight for each feature representative of its importance, it is best to standardize the features using a $z - score$. This will result in inputs to the model on the same scale. But to normalize the data, we should consider the test set as unseen. As a result, we need to learn the parameters required to standardize the data from the training set and use the same parameters to transform the test data.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
in_train, in_test, out_train, out_test =
train_test_split(input, outcome, test_size=0.20)
scaler = StandardScaler()
scaler.fit(in_train)
in_train = scaler.transform(in_train)
in_test = scaler.transform(in_test)
```

In the above code block, the "scaler.fit" step will learn the required parameters to standardize each feature separately. Then, using the learned parameter from the train set (in train), both the training and test datasets are transformed.

Now, we can train the model using a training set to predict the output of the test data and evaluate its accuracy.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
reg = LinearRegression().fit(in_train, out_train)
pred_test = reg.predict(in_test)
print(mean_squared_error(out_test, pred_test))
```

Here we used mean squared error to measure accuracy, but many other metrics can be used. The most popular ones that readers are encouraged to review are R square, mean absolute error, and mean relative absolute error.

Simple linear regression, as used above, does not always provide the best results possible. The outliers greatly impact it and may learn random variations within the training data, causing an issue known as model overfitting. The cost function can be adjusted to address these limitations by adding a penalization term to consider the features' weight. This will result in three derived regression models with differences in their regularization terms: Lasso, Ridge, and Elastic-Net.

In Lasso regression, weights will be used as the penalization term in the loss function formed as $\mathcal{L} = \sum \left(y_i - y_i' \right)^2 + \lambda \sum \# w_i \#$, where λ is a hyperparameter to tune and w_i is the weight for the feature i.Using the sum of the absolute values of weights, known as L1 loss, will result in smaller weights closer to zero. This will cause sparsity in the weights and is used as a feature selection method. Below are the codes needed to import and train Scikit-Learn's Lasso regression [4] to make predictions for the discussed dataset.

```
from sklearn.linear_model import Lasso
reg2 = Lasso().fit(in_train, out_train)
pred2_test = reg2.predict(in_test)
print(mean_squared_error(out_test, pred2_test))
```

Note 1 For simplicity, here we trained Lasso regression using the default parameters. However, given that we now have λ as a hyperparameter, we may need to tune it and find the best value for it. One approach would be to split the training dataset into new training and validation sets. Models can be trained on the new training set using different values of λ, and results will be evaluated on the validation set to find the best value. Then, the model with the best λ value can be used to evaluate the accuracy of the test dataset. This approach will allow us to fine-tune the model's hyperparameter and use the best model for the test dataset without data leakage.

Ridge regression will similarly penalize the weights of the features in the loss function. However, the final loss function will be in the form of $\mathcal{L} = \sum_i \left(y_i - y_i'\right)^2 + \lambda \sum_i w_i^2$. Using the sum of squares of weights in the loss function, known as L2 loss, will penalize the larger values with higher intensity, resulting in a more uniform range of weights. Ridge regression is useful when there are too many features or when features have a high degree of multicollinearity. The code below utilizes Ridge regression for modeling and prediction.

```
from sklearn.linear_model import Ridge
reg3 = Ridge().fit(in_train, out_train)
pred3_test = reg3.predict(in_test)
print(mean_squared_error(out_test, pred3_test))
```

Finally, Elastic-Net combines the two aforementioned regularization terms to form the loss function $\mathcal{L} = \sum_i \left(y_i - y_i'\right)^2 + \lambda_1 \sum_i \left|w_i\right|^2 + \lambda_2 \sum_i w_i^2$. Here, we can take benefit from both the regularization terms discussed for Lasso and Ridge regression. However, the hyperparameter turning for both λ_1 and λ_2 is needed.

```
from sklearn.linear_model import ElasticNet
reg4 = ElasticNet().fit(in_train, out_train)
pred4_test = reg4.predict(in_test)
print(mean_squared_error(out_test, pred4_test))
```

Note 2 There are different methods of hyperparameter tuning. Grid search will evaluate all possible combinations of hyperparameters with their provided search space to find the best subset. This search method implemented in Scikit-Learn [5] is recommended for a small set of hyperparameters. The search space will grow exponentially with an increase in the number of hyperparameters or their corresponding search space. Random search [6] implementation will use a random combination of values for each hyperparameter for a preset number of iterations and report the best combination. Although random search has gained a lot of interest and has proven to be efficient, more advanced techniques exist to better utilize any correlation between the accuracy achieved and the parameters tried to make a more informed decision about the next set of values to try. The HyperOpt [7] library is an example that utilizes Bayesian optimization for hyperparameter tuning.

2.2 Classification

In classification, the goal is to use ML algorithms and assign a class label to the input examples. For instance, class labels can be patients' death or discharge outcomes at the end of a treatment or negative or positive blood test results. These examples are called binary classification since one of the outcomes can happen at a time and can be coded as a zero or one output for a model. It is possible to have more than two outcome classes as well, where instances can still belong to only one class, usually referred to as multiclass classification. In the case of multilabel classification, input examples can have more than one class label.

There are many ML algorithms designed to address each task mentioned above. Some can only perform simple binary classification, while many are inherently capable of multiclass or multilabel classification. Still, advanced techniques can be incorporated to use a simple binary classifier and form a multiclass or multilabel classifier.

Logistic regression is a simple binary classifier that essentially passes the output of a linear regression model, here $f(x)$, through a sigmoid function $p(x') = 1/1 + e^{-(x')}$. This will result in an output with values between zero and one. The final class label can be considered a positive outcome for a predicted value above a specific threshold, while the value below the threshold will be interpreted as negative.

K-Nearest Neighbor (KNN) is a multiclass classifier. It uses a measure of distance[2] (e.g., Euclidean, Hamming, Cosine, Manhattan) to find the K closest neighboring data instances in the training set to a given input sample. Then, the class labels for the determined K-nearest neighbors will be used in a voting scheme to decide the best matching class label. While having a simple algorithm, KNN does not perform well on very large or high-dimensional datasets.[3]

In recent years, tree-based models have shown great efficiency for classification tasks with high prediction accuracy. Tree-based models are based on the simple idea of a decision tree, where a series of conditional steps are taken to make a decision. Figure 19.2 shows an example of a decision tree. However, simplicity comes with disadvantages. Model overfitting arises when the tree fits well to the training data but performs poorly on the testing set. Moreover, considering the order of the conditional steps taken, trees of various shapes will be generated and performed

[2] It is best to use a measure of distance relevant to and representative of available features.

[3] With an increase in data dimensionality, the data points will appear closer together, making this algorithm inefficient. Additionally, pairwise comparison to find the closest neighbors makes the algorithm in efficient in datasets with a large number of instances.

differently. These issues resulted in the invention of two well-known models based on decision trees: Random Forests (RF) and Gradient Boosting Decision Tresses (GBDT).

RF builds a bunch of decision trees independently, each making a simple prediction. Each tree's structure will be randomized and created on top of a bootstrap[4] sample of the training dataset. The final prediction by RF will be the aggregated prediction of all trees. The randomization of the structure and use of bootstrapping have made RF a powerful model resistant to outliers and missing values in the datasets.

Boosting

To build a more robust model, GBDT models and popular examples (e.g., XGBoost and CatBoost) use an ensemble of weak decision tree predictors. In GBDT, tresses are built iteratively. Meaning each tree is built after the other, and the previous step's output is used in addition to the features as input to the next tree. This will result in each new tree improving on the predictions made by the previous round, improving overall efficiency, a concept called boosting.

Let's get to coding by considering a classification task. For the dataset, we considered using Scikit-Learn breast cancer data [8]. This dataset includes samples of 569 patients with 30 numeric predictive attributes, each labeled as malignant or benign. Like regression, we need to load the dataset, split the data to train and test sets and standardize the features.

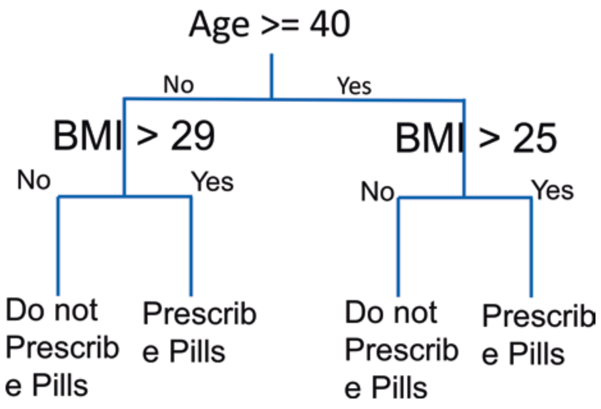

Fig. 19.2 A decision tree to predict patients' need for a prescription

[4] Random sampling with replacement.

```
breast_cancer = datasets.load_breast_cancer()
features = breast_cancer.data
labels = breast_cancer.target
f_train, f_test, l_train, l_test = train_test_split(features,
labels, test_size=0.20)
scaler = StandardScaler()
scaler.fit(f_train)
f_train = scaler.transform(f_train)
f_test = scaler.transform(f_test)
```

For the classification algorithm, we will use the Scikit-Learn implementation of RF [9]. We need to first fit the model to the training data using the ".fit" call. Then the trained model can be used to predict the class labels for the test dataset. Finally, to measure the model's performance, we can calculate the percentage of correct predictions using the "accuracy_score" function imported from "sklearn.metrics."

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
rf = RandomForestClassifier().fit(f_train, l_train)
pred_rf = rf.predict(f_test)
print(accuracy_score(l_test, pred_rf))
```

We can try different classification algorithms to find the best-performing algorithm for a given dataset. Here, we use the Scikit-Learn implementation of GBDT [10] to evaluate the performance gain achieved using a more advanced implementation of decision trees on the same predictive task.

```
from sklearn.ensemble import
GradientBoostingClassifier
gbc = GradientBoostingClassifier().fit(f_train, l_train)
pred_gbc = gbc.predict(f_test)
print(accuracy_score(l_test, pred_gbc))
```

Results show improved accuracy by using the GBDT model over the RF. However, it should be noted that there are hyperparameters for each algorithm that need to be tuned using grid search, random search, or Bayesian optimization techniques, as discussed previously.

Note 3 For RF, some important hyperparameters to consider are the number of estimators, the maximum depth of trees, and the criterion to measure the quality of splits for each feature. GBDT, while providing a similar hyperparameter to tune, has a few additional unique hyperparameters, such as loss function and learning rate, to consider.

Note 4 It should be noted that the percentage of correct predictions for a classification task is not the only measure of its performance. The confusion matrix, the area under ROC curve (AUC), and the F1 score are a few others.

2.3 Clustering

So far, all the ML algorithms we discussed are considered supervised learning techniques. In supervised learning, the data instances have a class label or a value assigned. As an expected outcome, this value or label will be used to train a model and later need to be predicted for new instances. In contrast, clustering is an unsupervised ML method that involves discovering a natural groping among the examples. A cluster is an area of densely populated samples or samples closer to each other. Clustering helps us better understand a problem or dataset, group similar data instances, or map new data to an existing cluster in the dataset. Figure 19.3 shows data instances in a two-dimensional space with three clusters. Each cluster is identified with a separate color surrounded by a black circle and a gray dot in the center as a cluster centroid.

K-means and OPTICS are two clustering algorithms we are going to discuss here. K-means requires the number of clusters (K) and a measure of distance to be defined and used for calculation. It starts by considering K random values[5] as cluster centroids. Then, it calculates the distance between each data instance and all the cluster centroids, assigning the closest cluster to it. When the initial clustering happens for all the data instances, K-means will move each cluster centroid to the center of the data instances assigned to it.[6] With the new cluster centroids, K-means will recalculate the distances and assign data instances to the newly formed cluster centers. This process will be repeated until there is no change in the assigned cluster for any data instances or it reaches an identified maximum number of iterations.

While K-means splits the feature space into distinct areas with its measure of distance, OPTICS uses a measure of density and reachability in a provided neighborhood for clustering. Although both clustering algorithms may provide the same results in some cases, the results of applying each clustering algorithm could be

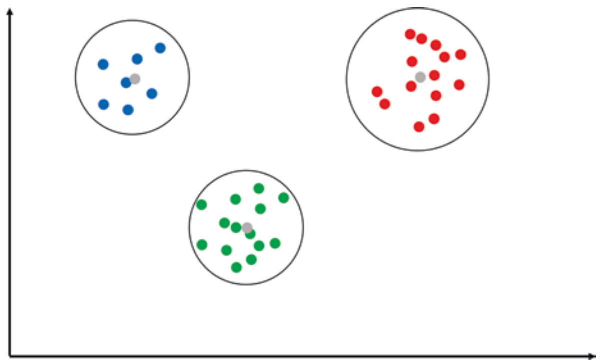

Fig. 19.3 Three clusters of data in a two-dimensional space

[5]The value considered as K is a hyperparameter that needs to be tuned.

[6]This will be done by taking the average of each feature for the data instances assigned to the same cluster.

drastically different. Many evaluation metrics are devised for clustering algorithms. However, there is no best or easiest method of comparison. Evaluation of the identified cluster may require controlled experiments or domain expert knowledge. Figure 19.4 shows clusters found by both K-means and OPTICS differentiated by their colors.

To increase the interpretability of our analysis, here we use the Scikit-Learn synthetic dataset generator to create a data set for clustering. The dataset consists of 500 samples in a two-dimensional space, intentionally generated to have five clusters with varied densities, as shown below.

```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=500, centers=5,
cluster_std =1.00)
```

We can now import K-means clustering [11] from Scikit-Learn and apply it to synthetically generate a dataset[7] to form five clusters as we expect. The clusters will be formed using the ".fit" call, and by ".cluster_centers_" we can print out the centers calculated.

```
from sklearn.cluster import KMeans
km = KMeans(n_clusters=5)
km.fit(X)
print(km.cluster_centers_)
```

We will use the Seaborn library to generate the figure for the clustered data. Figure 19.5 shows the clusters identified by different colors for the synthetically generated data with five known clusters.

```
import seaborn as sns
sns.scatterplot(x=X[:,0], y=X[:,1], c= km.labels_)
sns.scatterplot(x=km.cluster_centers_[:, 0],
y=km.cluster_centers_[:, 1], c=['black'])
```

Like K-means clustering, we can apply OPTICS [12] to the same dataset getting the identified clusters and comparing the two algorithms. Figure 19.6 shows the results of utilizing the OPTICS algorithm.

```
from sklearn.cluster import OPTICS
opt = OPTICS()
opt.fit(X)

sns.scatterplot(x=X[:,0], y=X[:,1], c= opt.labels_)
```

[7]Forming a train and test splits may not be required as we do not have any assigned labeled to the data.

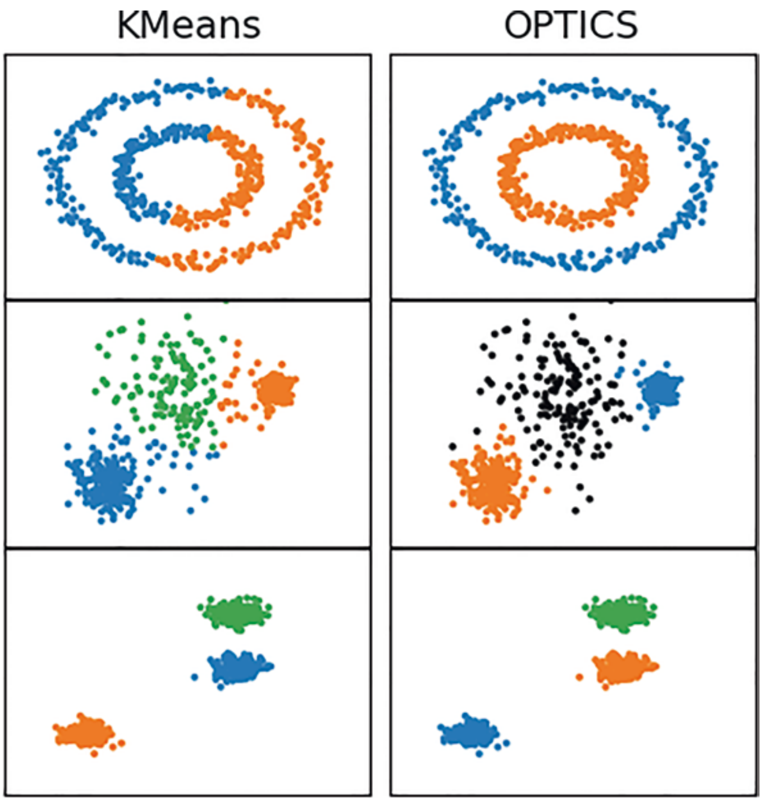

Fig. 19.4 Clusters found by both K-means and OPTICS

While the results for K-means clustering seem more appealing, if we consider the density-based nature of the OPTICS algorithm, we can conclude that both algorithms are clustering the data perfectly based on their similarity measure. We can observe that K-means separated the two-dimensional space into K = 5 distinct regions with data instances close together. At the same time, optics has found and clustered the data instances with the same density at the center and border of each cluster.

Note 5 Similar to supervised learning algorithms, unsupervised techniques also have hyperparameters that need to be tuned. For K-means, the number of clusters (K) is one of the most important parameters. K can be found using the elbow method. With the elbow method, clustering will be conducted using different K values, followed by the calculation of inertia.[8] The K value as a function of the number of clusters is best where the highest reduction in the inertial is observed, and

[8] Inertia measures the sum squared distance between each data point and its assigned centroid.

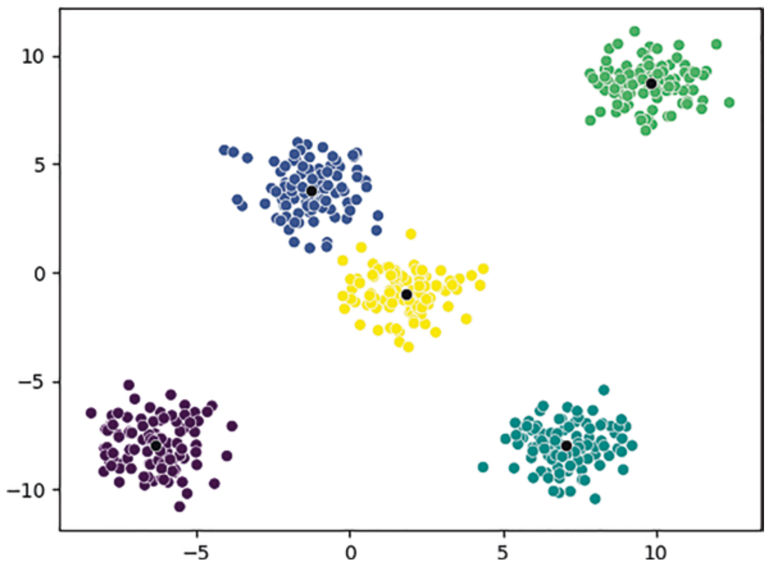

Fig. 19.5 K-means clustering and synthetic data

then a plateau is reached.[9] For OPTICS, minimum samples and maximum neighborhood distance are hyperparameters to consider. However, OPTICS presents less sensitivity to hyperparameters than its density-based clustering predecessor, DBSCAN [13].

Note 6 The silhouette score is another metric for evaluating the clustering algorithms. It measures the degree to which the data points are similar within the assigned cluster compared to neighboring clusters. The silhouette value ranges from -1 to 1, with 1 representing the best match and -1 representing the opposite.

3 Neural Networks and Deep Learning

The first interesting and practical demonstration of DL goes back to 1989, when Yann LeCun implemented a NN for handwritten digit recognition. But the capabilities are not limited to computer vision. Now, NNs are used for regression, time-series prediction, object detection and segmentation, robotics, self-driving cars, and even NLP to evaluate text sentiment or generate responses to a question.

NNs, or artificial NNs, are a subset of ML and the heart of DL, inspired by the human brain. It is composed of three or more layers (an input layer, one or more

[9]With an increase in the number of clusters, the inertia will constantly decrease, with the lowest inertia achieved with K equal to the number of data instances.

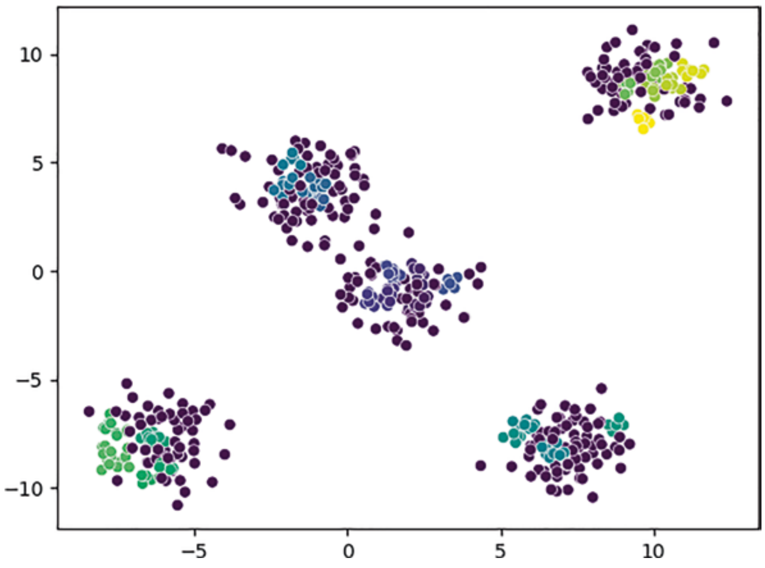

Fig. 19.6 OPTICS clustering and synthetic data

hidden layers, and an output layer). Each layer is composed of neurons connected to other layers with edges and their associated weights. If the sum of the inputs to a neuron, multiplied by their corresponding weights and added by the neuron's bias value, passes a threshold,[10] the neuron will be activated and pass the signals to the next layer. Generally, NNs start with random weights[11] and zero biases assigned to the edges and neurons. Optimization is needed to generate the required output model to adjust these values. Many efficient optimization algorithms[12] have been devised based on gradient descent and backpropagation, making efficient model adjustments possible. In this order, the inputs will be provided in a forward pass to the model to generate an output. Then the difference between the generated and expected output will be used in a backpropagation step by an optimization algorithm to adjust the weights and biases, reducing the model's error.[13] Figure 19.7 shows a NN with three neurons in the input layer, two hidden layers each with five neutrons, and an output layer with two neurons. Here, colored in blue, are the weights connecting input 1 to the next layer of neurons in hidden layer 1.

NNs with enough layers, the correct number of neurons, and the correct settings can virtually simulate any function and provide higher accuracy than traditional models. However, an increase in the model's complexity to achieve higher

[10] This threshold and the degree to which a neuron will be activated are defined by an activation function. The choice of activation function will influence the model's training time and accuracy.

[11] For further details, readers are encouraged to see He and Glorot weight initialization.

[12] Adam optimizer, RMSprop, and AdaGrad are a few to name.

[13] The model's error is calculated using a cost function.

accuracies comes at the cost of requiring more training data. NNs are also better at feature selection than traditional ML models and can accurately learn important features among abundant input. This section will use the Keras v2.10 [14] library backed by TensorFlow v2.8 [15, 16] to develop some of the architectures useful for supervised DL tasks.

3.1 NN for Regression

Let’s design the architecture of our model to solve the same regression problem we discussed in the first part of this chapter, the diabetes dataset. We have the data loaded, split into training and test sets, and normalized, ready to be used for training a predictive model.

After installing and importing TensorFlow and Keras into the environment, first, we need to specify the type of model implementation we want to use.[14] A sequential implementation is chosen here. Now we can add layers of NN one after the other using the “.add()” command, where the output of one layer will be automatically forwarded to the next layer as input. For the next couple of fully connected layers, a few parameters need to be set. We need to specify the input size for the first layer, the number of neurons, and the activation functions for each layer. We can observe in the code block below that we used the input size of 10, the same as the number of input features, the first and second hidden layers each with 20 neurons,[15] and the

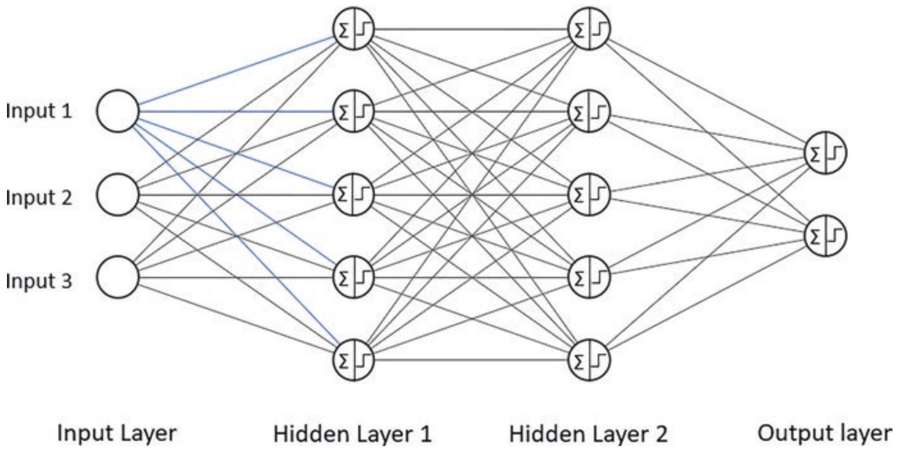

Fig. 19.7 A simple deep neural network

[14]There are three types of model implementation in TensorFlow: sequential, functional, and subclassing.

[15]The number of hidden layers and the neurons in each layer are hyperparameters that need to be tuned for each model and dataset.

final output layer of size 1 just to predict a single numerical value. The higher the number of neurons in each layer, the more connections with their associated parameters in the model that need to be tuned, increasing the chance of model overfitting. To solve this issue, more training data with a higher computation cost and training time may be required to achieve a good generalizable model.[16]

For each fully connected layer in our model, a few activation functions are available to choose from. Some that we can utilize here are Sigmoid, Tanh, Linear, and Relu. Here we used the Relu function for the first and second hidden layers, as it has proven to be suitable for many predictive tasks and helps train and converge the model faster with fewer epochs.[17] However, the activation function for the output layer is chosen to be a linear function, consistent with the regression task and the range of values to be predicted.

```
from tensorflow import keras

model = keras.models.Sequential()
model.add(keras.layers.Dense(20, input_dim=10,
activation="relu"))
model.add(keras.layers.Dense(20, activation="relu"))
model.add(keras.layers.Dense(1, activation="linear"))
```

The "model.summary()" helps us get a summary of the defined model with the number of trainable parameters within each layer. By compiling the model, we are required to specify a loss (cost) function, an optimizer algorithm, and an accuracy metric. The loss function provides the library with a means to evaluate how much the generated output deviates from the expected output in each training epoch. And the optimizer algorithm tries to minimize the value of the loss function by adjusting the model's parameters in backpropagation steps. The Adam optimizer is used here due to its accuracy and efficiency. The metrics provided will be used to evaluate the model's accuracy during training.[18] Finally, the model can be trained for the specified number of epochs by providing the training data and corresponding expected outputs. After completion of the training, the trained model's accuracy is evaluated by predicting the results for the test dataset.

[16] To prevent overfitting and get a good generalization, the use of the "Dropout" layer is recommended. This technique will randomly deactivate some of the neurons in each training epoch.

[17] Each forward pass to feed the data into the model, getting an output, and its corresponding backpropagation pass to adjust the model are called an epoch.

[18] While in this task, the loss (cost) function and accuracy metric are the same, this is not always the case and usually happens for regression models.

```
model.summary()

model.compile(loss= "mean_squared_error" ,
optimizer="adam", metrics=["mean_squared_error"])
model.fit(in_train, out_train, epochs=20)

pred_test= model.predict(in_test)
print(np.sqrt(mean_squared_error(out_test,pred_test)))
```

Note 7 One of the most important hyperparameters in NN is the number of layers and neutrons in each layer. These parameters should be tuned for each predictive task accordingly. Grid Search, Random Search, or Bayes Optimization can be utilized.

Note 8 While we used well-known metrics of accuracy, optimization algorithm, and cost function, there are many others to try, affecting the final model's accuracy. It is advised to select a few different options and consider experimenting with a number of combinations to achieve the best possible results.

Note 9 NNs use random weights for model initialization. It is best to make the results replicable by setting a seed value[19] for the random number generator in both the TensorFlow and NumPy libraries.

3.2 NN for Classification

We will consider the breast cancer dataset used previously for the classification task. Here, the number of neurons in the last layer should equal the number of classes in the dataset. As a result, each neuron corresponds to one class. During model training and prediction, the neuron in the last layer generating the largest output will be considered the final predicted class label. However, for a binary classification (cancer or not cancer), it is possible to use a single neuron at the output layer. With a single neuron, results can be interpreted as positive if the generated output is above a threshold and negative in reverse. To practice multiclass classification, we will consider two output neurons to simulate a scenario that can be extended to more than two classes.

We use the sequential model implementation here. There are 30 features in the dataset used as inputs to the model (input_dim). It is recommended to have more neutrons than inputs in the subsequent layers, with two layers of each 50 neurons.

[19] Seed is the value used in computer systems to generate a sequence of pseudo random numbers. By providing an initial input (seed) to a random number generator, the same sequence of random numbers will be generated.

Fully connected layers may overfit, memorizing the random variations in the data and noise instead of learning interactions. Randomly disconnecting some edges between the fully connected layers will improve the model's generalizability. This is achieved using dropout layers after each fully connected layer. In the last layer, we use two neurons equal to the number of classes we are predicting. Moreover, the SoftMax activation function is considered for the last layer, making the summation of all output equal to one[20] and simulating an Argmax Function.[21]

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(50, input_dim=30,
activation="relu"))
model.add(keras.layers.Dropout(0.25))
model.add(keras.layers.Dense(50, activation="relu"))
model.add(keras.layers.Dropout(0.25))
model.add(keras.layers.Dense(2, activation="softmax"))
model.summary()
```

To compile the model for classification, we need to use categorical cross-entropy loss [17]. Moreover, we considered categorical accuracy to compare if the class with the highest predicted probability matches the label provided in the dataset.

```
from sklearn import preprocessing

model.compile(loss= "categorical_crossentropy" ,
optimizer="adam", metrics=["categorical_accuracy"])
```

The model here has two output neurons, and TensorFlow expects the class labels in a OneHot-encoded format for multiclass classification. For each class, we need to have a column, and only one column per data instance can have one value, representing the corresponding class label. The code below will apply the required transformation to both training and test labels.

```
lb = preprocessing.OneHotEncoder(sparse=False)
lb.fit(l_train.reshape([-1,1]))
binerized_l_train = lb.transform(l_train.reshape([-1,1]))
binerized_l_test = lb.transform(l_test.reshape([-1,1]))
```

Now, we need to fit the model with training data.[22]

[20] Converting numbers into a predicted probability distribution.

[21] Argmax function sets the largest predicted probability equal to one and the remaining values equal to zero.

[22] More training epochs are considered here compared to the previous example with more neurons. The number of training epochs is a parameter that needs tuning and should be chosen by considering the model's training loss to prevent overfitting.

```
model.fit(f_train, binerized_l_train, epochs=50)
```

For the final model evaluation, we will predict the test set's class labels and compare them against the actual labels. As a measure of accuracy, we will calculate the percentage of correct predictions using the Scikit-Learn "accuracy_score" function.[23] One important note is that the model's predictions are probabilities for each class. To convert these to actual class labels, we need to consider the class with the highest probability for each instance as the final prediction. The NumPy Argmax function will look at the predicted probabilities for each instance and return the corresponding column number for which it has the highest probability.

```
from sklearn.metrics import accuracy_score

pred_prob_test= model.predict(f_test)
prediction_test =  np.argmax(pred_prob_test, axis=1)
print(accuracy_score(l_test, prediction_test))
```

Note 10 In this example, all the features are numerical values. Category features should be converted to OneHot-encoded versions before being used as input to the model.

3.3 NN for Computer Vision

The field of computer vision is interwoven with Convolutional Neural Networks (CNNs). CNNs have helped computers achieve accuracy above humans in visual tasks. CNNs are used in many tasks, from object detection and classification to self-driving cars and, recently, in many medical imaging domains to detect symptoms and automate the diagnosis process. But what are the CNNs and how a simple CNN can be implemented for a computer vision task are what we will discuss in this section.

In fully connected NNs, each neuron is connected to every neuron in the next layer. This architecture causes a few issues in image processing. If you use fully connected layers for a computer vision task with an image as an input, while the model may learn and predict the assigned classes, shifting or scaling the object in the image would easily affect the final prediction. Since the model has memorized the exact location and size of an object. To address these limitations, we need feature extractors that can learn simple features or patterns in the initial layers of NNs. More complex features and combinations are learned as the data moves along the layers. The first few layers may only learn the horizontal, vertical, and diagonal lines; while moving along the layers, the combination of features in the previous

[23] Other metrics to name are the confusion matrix or area under receiver-operating characteristics.

layer can be learned to shape more complex patterns. Generally, a convolution in CNN acts the same as a feature extractor. It will shift over the image and extract features by multiplying with the underlying pixels or values in each step. Figure 19.8 shows an example of a convolutional feature extractor (filter) in CNN. The pixel[24] values for a hypothetical black-and-white[25] image of size 5 by 5 are shown in white with a feature extractor of size 3 by 3 in green. The feature extractor moves over the image calculating the summation of the products. The resulting output will be of size 3 by 3, being forwarded to the next layer.

Convolutional layers are usually followed by a pooling layer, reducing the output size and decreasing the number of parameters required to train in the subsequent layers. A very popular pooling layer is max pooling of size of 2 by 2, convolving similarly over an image, taking the max value of the pixels. Figure 19.9 shows an example of a max pooling filter applied to the final results of Fig. 19.8.

Now that we know the basics of CNNs, let's get to coding and see how we can use layers of CNNs to detect patterns within an image.

Fashion_mnist is a dataset of images with 10 classes of clothing items. Each image has one layer, as images are black and white. The first step is to load the images from the dataset and reshape them into arrays with four dimensions, as TensorFlow requires. In the reshape function, we can see the requested dimensions in order: the number of training samples (60,000), the height and width of each image (28 × 28), and the number of layers in each image [1]. To normalize the data for image processing, all the values can be divided by the maximum intensity (255), resulting in an intensity value of 0–1. We need to repeat the same process for our test dataset as well.

```
import tensorflow as tf

mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (test_images,
test_labels) = mnist.load_data()

training_images = training_images.reshape(60000, 28,
28, 1)
training_images = training_images / 255.0
test_images = test_images.reshape(10000, 28, 28, 1)
test_images = test_images / 255.0
```

After loading the data, it is time to build our model. Using TensorFlow sequential modeling, we provide the layers in order. In the first layer, we want 64 convolutions with a filter size of 5 by 5 and a ReLu activation function. The input size would be the same as the size of each image, 28 by 28, with 1 layer. The output of this layer

[24] The basic unit of a digital image that can be displayed on a digital device or display.

[25] Black and white images can be represented by a single layer, with values presenting the intensity of the light.

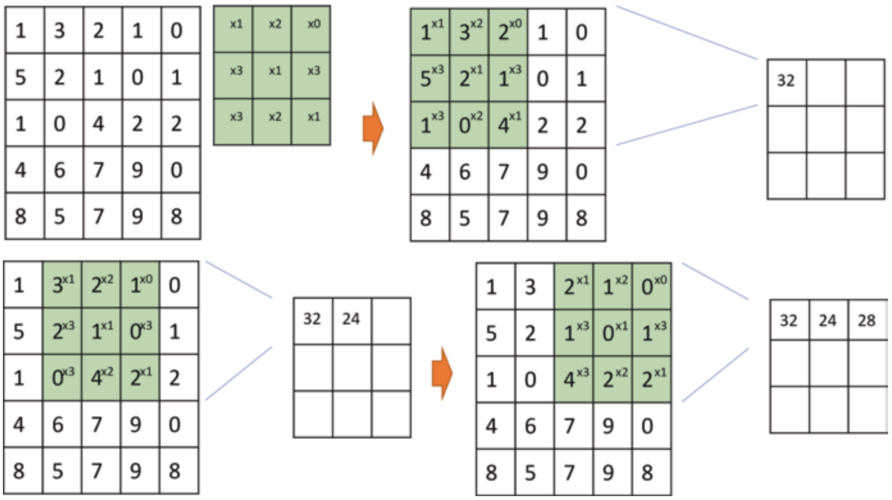

Fig. 19.8 A single convolutional filter convolving over an image

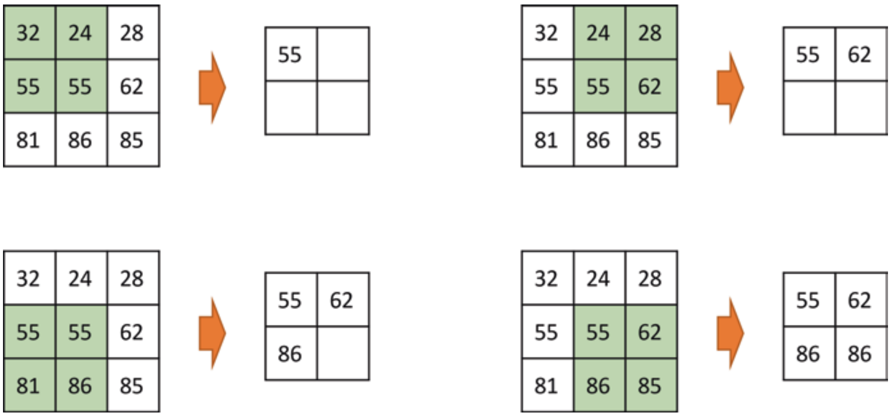

Fig. 19.9 A max pooling filter convolving over an image

will automatically be forwarded to the next layer. After the convolutional layer, we utilize a pooling layer to reduce the dimensions of the features. We have one more layer of convolutions with 64 filters of size 3 by 3, followed by a pooling layer of size 2 by 2. These layers will extract features from the images. Now, we need to flatten the multidimensional output to one dimension using the "flatten()" function and send it to two fully connected layers for final classification. The final layer needs to have 10 neurons, equal to the number of classes.

```
model = tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(64,(5,5), activation ='relu',
input_shape=(28,28,1)),
        tf.keras.layers.MaxPooling2D(2,2),
        tf.keras.layers.Conv2D(64, (3, 3),
activation='relu'),
        tf.keras.layers.MaxPooling2D(2, 2),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation=tf.nn.relu),
        tf.keras.layers.Dense(10, activation=tf.nn.softmax)
   ])
```

To compile the model we just defined, we need to specify the optimizer algorithm, loss function,[26] and accuracy metric. Then, we can get the model summary for parameters and layers and start the training.

```
model.compile(optimizer = 'Adam',
            loss = 'sparse_categorical_crossentropy',
            metrics=['accuracy'])
model.summary()
model.fit(training_images, training_labels, epochs=5)
```

With the trained model, now we can predict the test set and evaluate the accuracy of the model.

```
model.evaluate(test_images, test_labels)
```

3.4 NN for NLP

NLP is a branch of ML that allows computers to process and understand text data. This includes, but is not limited to, sentiment analysis, question answering, text summarization, and machine translation. In recent years, DL has revolutionized NLP by developing language models capable of understanding context, trained on millions of documents. In this section, as an introduction to NLP, we will develop a model for sentiment analysis, predicting the positivity and negativity of a text written by a user as a review.

Like many other NLP tasks, the first step is data cleaning and selecting a method for sentence representation. Data cleaning (depending on the task) refers to removing punctuation, extra spaces, HTML tags in the text, emojis, hyperlinks, and stop words. There are quite a few approaches to representing the sentences. One of the

[26] The use of sparse categorial cross-entropy loss would eliminate the need to provide the one hot encoded version of class labels.

old methods is a one-hot representation, with one column per word in the dataset. If a word appeared in the sentence, its corresponding column would have a value of 1 and otherwise be zero. While this helps to capture sentiments and represent words in a sentence, the words' order and appearance together will be removed, resulting in information being lost. Using bi-gram and tri-gram can help capture words that appear together and in sequential order. However, this method also suffers from not utilizing the context in which the words have appeared.

For better representations, instead of a sparse matrix with columns representing the words, using an n-dimensional vector to represent each word is a better approach. This n-dimensional representation should encode the word meaning, referred to as embedding. With this approach, we can replace words with embeddings and represent sentences with word-length sequences. There are many pre-trained embeddings[27] to utilize. However, embeddings can be directly learned from a dataset as well.

The dataset we will use here has words pre-converted to unique integer values for simplicity. Integer representations are not recommended as they are randomly chosen. Using an embedding layer in TensorFlow, we can learn the best representation from the dataset for the integer representation provided here.

First, we load the dataset from TensorFlow. "Num_words" limits the TensorFlow to represent the top 10,000 words based on the frequency of their repetition, considering others as unknown (oov_char).[28] "Maxlen" instructs the library to truncate any sentence longer than 512 to the same size. Shorter sentences will be padded by zero, making all the representations the same length.

```
import tensorflow as tf

(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.imdb.load_data(num_words=10000)

x_train =
tf.keras.preprocessing.sequence.pad_sequences(x_train,
maxlen=512)
x_test =
tf.keras.preprocessing.sequence.pad_sequences(x_test,
maxlen=512)
```

Now we need to define our network. The "input" variable is a placeholder representative of the sentence that will be passed to the model. The first layer would be the Embedding layer. TensorFlow has recently added this capability, where the model can learn the best representation of the words based on the input dataset. This helps to use representations that accurately encode the word's meaning instead of a randomly generated integer by the data loader. The first value (10000) provided to

[27] Glove and word2vec are two of the statically generated word embeddings.

[28] This helps to remove the words that have a very low frequency of repetition.

the embedding layer will represent the expected number of vocabulary words, and the second is the number of output dimensions for representing each word.

```
inputs = tf.keras.Input(shape=(None,), dtype="int32")
x = tf.keras.layers.Embedding(10000, 100)(inputs)
```

We need NN models capable of understanding the sequential nature of words in a language. Although using fully connected layers may be possible, they could not understand the context in which the words appeared accurately. Using recurrent NNs (RNNs), not only are the neurons in each layer connected to the next layer, but there is also a hidden state connecting the neurons within the same layer. These connections are usually unidirectional.[29] But the unidirectional connections can utilize the context information of the previous words for representation. Here, we use a bidirectional implementation of the RNN called Long Short-Term Memory (LSTM) networks to address this limitation. The bidirectional implementation helps information flow from the beginning to the end of a sentence and in reverse, while the LSTM model helps better understand long sequences of words. Finally, the last layer has a neuron as a final classifier, predicting the sentiment for this binary classification task.

```
x = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64,
return_sequences=True))(x)
x =
tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64))(x)
outputs = tf.keras.layers.Dense(1,
activation="sigmoid")(x)
```

The final step would be setting the input and output of the model, followed by compilation and evaluating the model's performance.

```
model.compile("adam", "binary_crossentropy",
metrics=["accuracy"])
model.fit(x_train, y_train, batch_size=64, epochs=3) # ,
validation_data=(x_test, y_test)

model.evaluate(x_test, y_test)
```

[29] From the first neural to the last, or from the beginning of the sentence to the end of the sentence.

4 Recent Advancements

While here, we provided a couple of source codes and discussed the implementation details. Examples are countless. Applications of ML and DL are not limited to regression and classification tasks or sentiment analysis. In computer vision, with the help of DL, machines can find an object of interest within an image (object detection) and provide an accurate bounding box for it (object localization). They are helping automate the disease diagnosis processes and assisting doctors by proposing areas that might need more attention. Using image segmentation techniques in computer vision, we can provide an exact area where an abnormality is observed with pixel-level accuracy (object segmentation). Studies now focus on utilizing DL to increase the quality of MRI and X-ray images. In NLP, we used the patients' historical notes in EHR to provide clinicians with a summary or even propose a discharge summary. They are helping to improve clinical documentation and saving clinicians' time.

5 Models' Interpretability

A black-box ML model usually focuses on predicting outcomes, but little insight is available beyond the predictions. However, recent years have witnessed numerous advances in producing robust and interpretable insights from complex machine-learning models. Some of the examples are the Grad-CAM [18, 19], LIME [20, 21], and SHAP [22, 23] libraries utilized in many applications. The most popular library, SHapley Additive exPlanation (SHAP), is based on the game theoretic approach, which has gained a lot of attention in many domains. SHAP provides insightful interpretations of a complex ML model with high accuracy and robustness, close to human interpretations. The generated SHAP values for input features of an ML model can be used to assess the effect of the inputs on the final model's prediction. There are abundant examples of applications in many domains, including tabular data modeling, text classification, question answering, image processing, and genomics.

6 Further Practice

1. What is the cost function?
2. Why do we need to split the data into train and test datasets?
3. Why do we need to standardize the input data?
4. What are the metrics to evaluate the accuracy of the linear regression model?
5. What are the differences between Lasso and Ridge linear regressions?
6. Why do we need to tune the hyperparameters on a validation set?

7. Which terms are different in the L1 and L2 losses?
8. What are binary, multiclass, and multilabel classifications?
9. What are RF and GBDT models, and how do they differ from Decision Trees?
10. What are the differences between K-means and OPTICS clustering algorithms?
11. What should be the input and output sizes of a NN for a given predictive task?
12. How the model generalization can be improved in fully connected NNs?
13. What should be the order of dimensions as input for image processing in TF?
14. Why do we need to use a pooling layer in CNN?
15. What are pre-trained embeddings, and how they can be utilized for an NLP task?

Answer Keys

1. A function that determines how well an ML model is performing
2. Because the model will be evaluated on the same data it has seen during training. As a result, an evaluation would not be representative of real-world performance with unseen data.
3. To have the same scale for all the data. Many ML algorithms are sensitive to the data scale and may find unrealized coefficients.
4. Mean squared error, mean absolute error, and *R*-squared.
5. Lasso cost function will result in some coefficient closer to zero and act as a feature selector, while Ridge will result in a coefficient more uniform and works better with multicollinearity.
6. Before applying models to the actual test dataset and evaluating the performance, hyperparameters should be tuned on an unseen part of the dataset. But this could not be the test set. This is why the training dataset is usually utilized to drive another subset of data to test the model with different hyperparameters to find the best, called the validation set.
7. L1 loss uses the sum of the absolute value of the weight for penalization, while L2 loss uses the sum of squared weights.
8. Binary classification refers to a task that can be seen as a binary outcome positive and negative. Multiclass classification has multiple outcomes in which only one can be true for each instance, while in multilabel classification multiple class labels can be assigned to a single instance.
9. RF uses an ensemble of Decision Trees (DT) using a bagging method by data resampling and combines the result using a voting method. GBDT uses sample weighting or output of previous models to build the next level of predictors, improving modeling accuracy interactively.
10. K-means used a method of distance to find K closes neighbor and uses voting for the final decision, while OPTICS uses density as a measure of clustering using minimum points in a neighborhood in a core distance.
11. Input should be equal to the number of features, size of an image, or length of a sentence based on a task. While output size is equal to the number of classes to predict.
12. Using the dropout layer model generalization can be improved.
13. First is the number of samples, then the size of each sample, and the last dimension represents the number of color channels.

14. The pooling layer reduces the dimensionality of feature, reducing the required computations
15. Pre-trained embeddings provide the representation for each word with meaning encoded in it. Each word should be replaced by a corresponding embedding representation before being forwarded to the model.

References

1. https://www.python.org/
2. https://scikit-learn.org/stable/
3. https://scikit-learn.org/stable/datasets/toy_dataset.html.org
4. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html.org
5. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html.org
6. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html.org
7. http://hyperopt.github.io/hyperopt/
8. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html.org
9. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.org
10. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html.org
11. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.org
12. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.OPTICS.html.org
13. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html.org
14. https://keras.io/getting_started/faq/
15. https://www.tensorflow.org/
16. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, et al. TensorFlow: a system for large-scale machine learning. In: 12th USENIX symposium on operating systems design and implementation (OSDI 16). 2016. p. 265–83. Available from: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf
17. https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy
18. Selvaraju RR, Cogswell M, Das A, Vedantam R, Parikh D, Batra D. Grad-CAM: visual explanations from deep networks via gradient-based localization. Int J Comput Vis. 2020;128(2):336–59.
19. http://gradcam.cloudcv.org/
20. https://homes.cs.washington.edu/~marcotcr/blog/lime/
21. Ribeiro MT, Singh S, Guestrin C. "Why should I trust you?": explaining the predictions of any classifier. arXiv; 2016. Available from: http://arxiv.org/abs/1602.04938
22. https://shap.readthedocs.io/en/latest/index.html
23. Lundberg SM, Lee SI. A unified approach to interpreting model predictions. In: Guyon I, Luxburg UV, Bengio S, Wallach H, Fergus R, Vishwanathan S, et al., editors. Advances in neural information processing systems. Curran Associates, Inc.; 2017.