1(a) Explain Supervised Learning and Unsupervised Learning

Supervised learning

In *supervised learning*, the training set you feed to the algorithm includes the desired solutions, called *labels* (Figure 1-5).

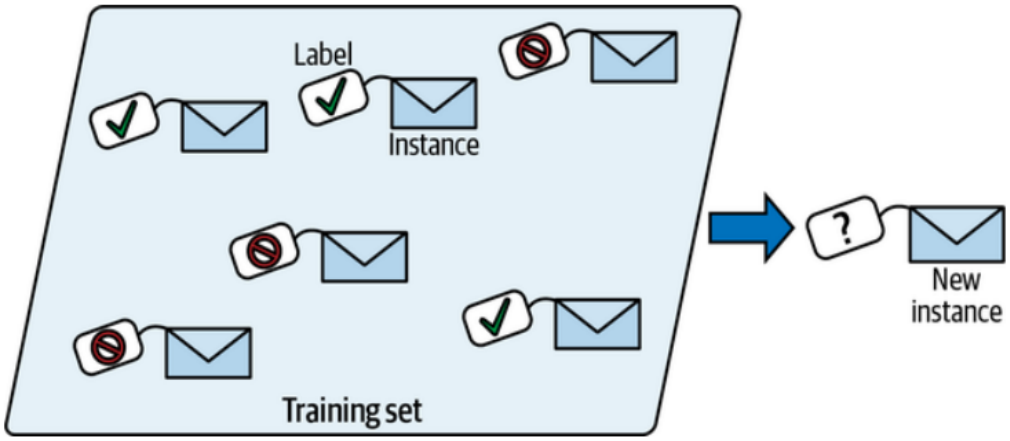


Figure 1-5. A labeled training set for spam classification (an example of supervised learning)

A typical supervised learning task is *classification*. The spam filter is a good example of this: it is trained with many example emails along with their *class* (spam or ham), and it must learn how to classify new emails.

Another typical task is to predict a *target* numeric value, such as the price of a car, given a set of *features* (mileage, age, brand, etc.). This sort of task is called *regression* (Figure 1-6).[1] To train the system, you need to give it many examples of cars, including both their features and their targets (i.e., their prices).

Unsupervised learning

In *unsupervised learning*, as you might guess, the training data is unlabeled (Figure 1-7). The system tries to learn without a teacher.

For example, say you have a lot of data about your blog's visitors. You may want to run a *clustering* algorithm to try to detect groups of similar visitors (Figure 1-8). At no point do you tell the algorithm which group a visitor belongs to: it finds those connections without your help. For example, it

might notice that 40% of your visitors are teenagers who love comic books and generally read your blog after school, while 20% are adults who enjoy sci-fi and who visit during the weekends. If you use a *hierarchical clustering* algorithm, it may also subdivide each group into smaller groups. This may help you target your posts for each group.

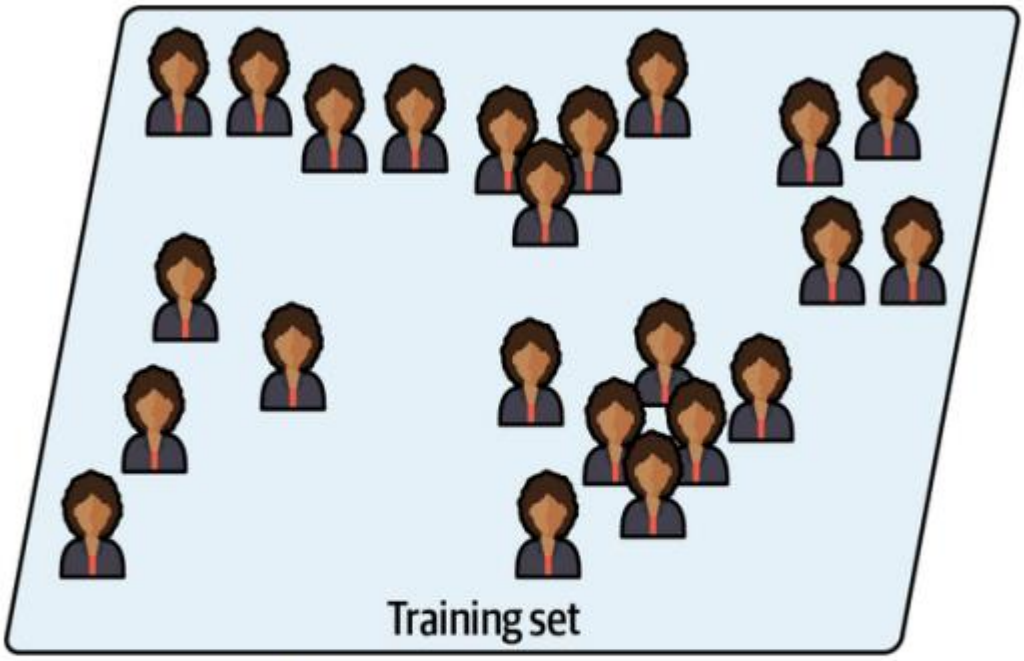


Figure 1-7. An unlabeled training set for unsupervised learning

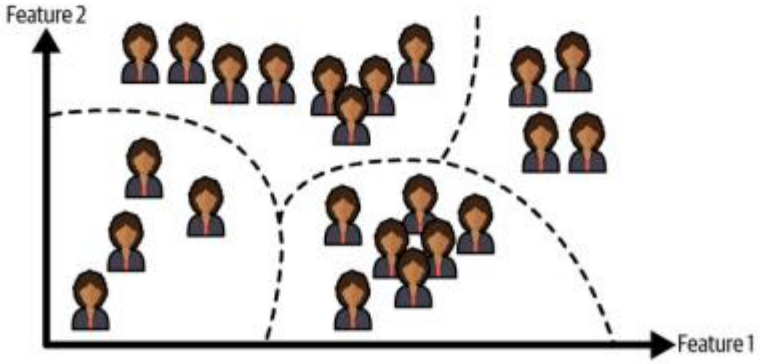


Figure 1-8. Clustering

Visualization algorithms are also good examples of unsupervised learning: you feed them a lot of complex and unlabeled data, and they output a 2D or

3D representation of your data that can easily be plotted (Figure 1-9). These algorithms try to preserve as much structure as they can (e.g., trying to keep separate clusters in the input space from overlapping in the visualization) so that you can understand how the data is organized and perhaps identify unsuspected patterns.

A related task is *dimensionality reduction*, in which the goal is to simplify the data without losing too much information. One way to do this is to merge several correlated features into one. For example, a car's mileage may be strongly correlated with its age, so the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear. This is called *feature extraction*.

Yet another important unsupervised task is *anomaly detection*—for example, detecting unusual credit card transactions to prevent fraud, catching manufacturing defects, or automatically removing outliers from a dataset before feeding it to another learning algorithm. The system is shown mostly normal instances during training, so it learns to recognize them; then, when it sees a new instance, it can tell whether it looks like a normal one or whether it is likely an anomaly (see Figure 1-10). A very similar task is *novelty detection*: it aims to detect new instances that look different from all instances in the training set. This requires having a very "clean" training set, devoid of any instance that you would like the algorithm to detect. For example, if you have thousands of pictures of dogs, and 1% of these pictures represent Chihuahuas, then a novelty detection algorithm should not treat new pictures of Chihuahuas as novelties. On the other hand, anomaly detection algorithms may consider these dogs as so rare and so different from other dogs that they would likely classify them as anomalies (no offense to Chihuahuas).

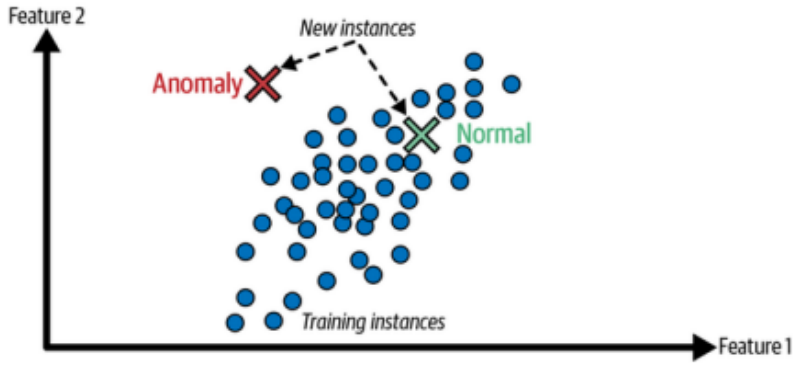


Figure 1-10. Anomaly detection

Finally, another common unsupervised task is *association rule learning*, in which the goal is to dig into large amounts of data and discover interesting relations between attributes. For example, suppose you own a supermarket. Running an association rule on your sales logs may reveal that people who purchase barbecue sauce and potato chips also tend to buy steak. Thus, you may want to place these items close to one another.

1(b) Outline the concepts involved in the following in context of

preparing the data for Machine Learning algorithms. Write code-snippet as applicable.

i. Data Cleaning,

ii. Handling text and categorical attributes

iii. Feature scaling

Prepare the Data

Notes:

- Work on copies of the data (keep the original dataset intact).
- Write functions for all data transformations you apply, for five reasons:
  - So you can easily prepare the data the next time you get a fresh dataset
  - So you can apply these transformations in future projects
  - To clean and prepare the test set
  - To clean and prepare new data instances once your solution is live
  - To make it easy to treat your preparation choices as hyperparameters

1. Clean the data:
   - Fix or remove outliers (optional).
   - Fill in missing values (e.g., with zero, mean, median…) or drop their rows (or columns).
2. Perform feature selection (optional):
   - Drop the attributes that provide no useful information for the task.
3. Perform feature engineering, where appropriate:
   - Discretize continuous features.
   - Decompose features (e.g., categorical, date/time, etc.).
   - Add promising transformations of features (e.g., log(x), sqrt(x), x^2, etc.).
   - Aggregate features into promising new features.
4. Perform feature scaling:
   - Standardize or normalize features.

In the context of preparing data for Machine Learning algorithms, the following concepts are essential:

i. Data Cleaning:

Data cleaning involves handling missing values, removing duplicates, and dealing with outliers in the dataset. It is crucial to ensure the data is accurate and reliable for training machine learning models. Here is a code snippet demonstrating data cleaning using pandas in Python:

```python
import pandas as pd

# Load the dataset
df = pd.read_csv('dataset.csv')

# Handling missing values
df.dropna(inplace=True)

# Removing duplicates
df.drop_duplicates(inplace=True)

# Dealing with outliers
# Define a function to remove outliers using z-score
def remove_outliers(df, column):
    z_scores = (df[column] - df[column].mean()) / df[column].std()
    df = df[(z_scores < 3) & (z_scores > -3)]
    return df

df = remove_outliers(df, 'feature_column')
```

ii. Handling text and categorical attributes:

Text and categorical attributes need to be converted into numerical representations for machine learning algorithms to process them effectively. This can be done using techniques like one-hot encoding or label encoding. Here is an example code snippet using pandas and scikit-learn for one-hot encoding:

```python
```

```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder

# Load the dataset
df = pd.read_csv('dataset.csv')

# Perform one-hot encoding on categorical column 'category'
encoded_df = pd.get_dummies(df, columns=['category'])

# Alternatively, you can use scikit-learn's OneHotEncoder
encoder = OneHotEncoder()
encoded_features = encoder.fit_transform(df[['category']]).toarray()
```

iii. Feature scaling:

Feature scaling is essential to ensure all features have the same scale, preventing certain features from dominating the model training process. Common techniques include standardization (scaling features to have zero mean and unit variance) or normalization (scaling features to a range between 0 and 1). Here is a code snippet using scikit-learn for feature scaling:

```python
from sklearn.preprocessing import StandardScaler

# Initialize the StandardScaler
scaler = StandardScaler()

# Fit and transform the features
scaled_features = scaler.fit_transform(df[['feature1', 'feature2']])
```

By incorporating these concepts into the data preparation process, you can ensure that your data is clean, appropriately encoded, and scaled for training machine learning models effectively.

1(c) Identify why model evaluation using k-Fold Cross Validation is

considered a better alternative over standard one set of train and

validation set.

Model evaluation using k-Fold Cross Validation is considered a better alternative over a standard one set of train and validation set for the following reasons:

1. **Better Utilization of Data**: In k-Fold Cross Validation, the dataset is divided into k subsets (folds), and the model is trained and evaluated k times, using each fold as a validation set once and the remaining folds as the training set. This ensures that each data point is used for validation exactly once, leading to better utilization of the available data.

2. **Reduced Variance**: By averaging the evaluation metrics over k iterations, k-Fold Cross Validation provides a more reliable estimate of model performance compared to a single train-validation split. This helps in reducing the variance of the evaluation metrics and provides a more stable assessment of the model's generalization performance.

3. **Mitigating Overfitting**: Using multiple validation sets in k-Fold Cross Validation helps in mitigating the risk of overfitting to a specific validation set. The model is evaluated on different subsets of data, which can help in identifying whether the model is generalizing well across different data distributions.

4. **Robustness**: k-Fold Cross Validation provides a more robust evaluation of the model's performance as it is less sensitive to the randomness in the data split compared to a single train-validation split. This robustness leads to a more reliable assessment of the model's ability to generalize to unseen data.

5. **Hyperparameter Tuning**: k-Fold Cross Validation is commonly used in hyperparameter tuning processes, such as grid search or random search, to find the optimal hyperparameters for the model. By evaluating the model on multiple folds, it helps in selecting hyperparameters that perform well across different data subsets.

Overall, k-Fold Cross Validation is preferred over a standard one set of train and validation set due to its ability to provide a more reliable estimate of model performance, better data utilization, reduced variance, and robustness in evaluating the model's generalization capabilities.

2(a) Explain any four of the main challenges of machine learning.

Main Challenges of Machine Learning

- Insufficient quantity of Training data
- Non representative Training data
- Irrelevant features
- Overfitting the Training data
- Underfitting the training data

Main Challenges of Machine Learning

In short, since your main task is to select a model and train it on some data, the two things that can go wrong are "bad model" and "bad data". Let's start with examples of bad data.

Insufficient Quantity of Training Data

For a toddler to learn what an apple is, all it takes is for you to point to an apple and say "apple" (possibly repeating this procedure a few times). Now the child is able to recognize apples in all sorts of colors and shapes. Genius.

Machine learning is not quite there yet; it takes a lot of data for most machine learning algorithms to work properly. Even for very simple problems you typically need thousands of examples, and for complex problems such as image or speech recognition you may need millions of examples (unless you can reuse parts of an existing model).

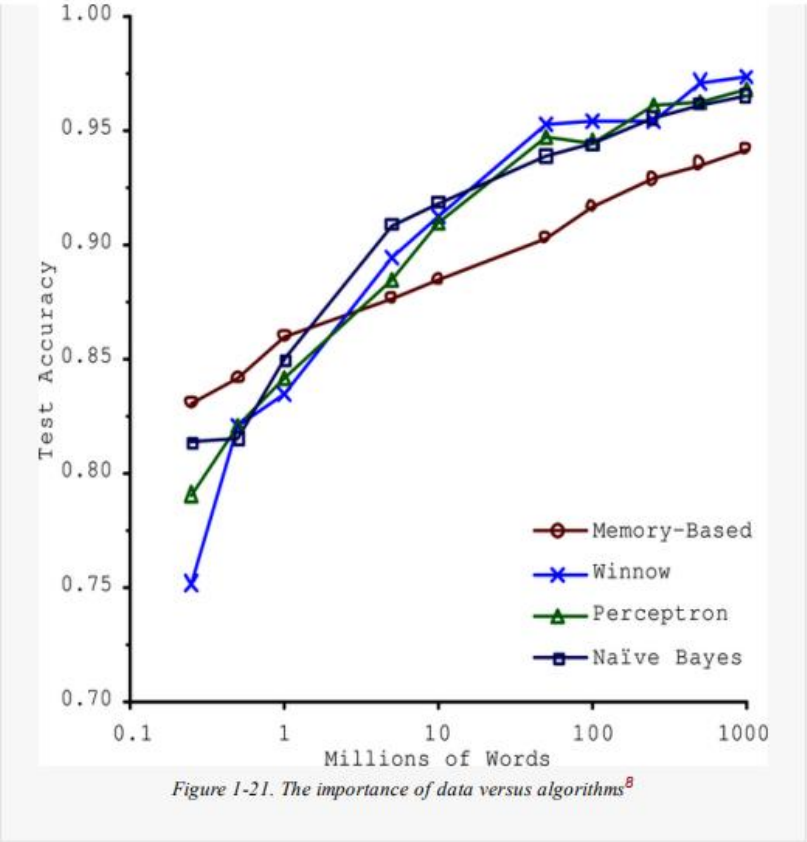


Figure 1-21. The importance of data versus algorithms[8]

Nonrepresentative Training Data

In order to generalize well, it is crucial that your training data be representative of the new cases you want to generalize to. This is true whether you use instance-based learning or model-based learning.

For example, the set of countries you used earlier for training the linear model was not perfectly representative; it did not contain any country with a

---

GDP per capita lower than $23,500 or higher than $62,500. Figure 1-22 shows what the data looks like when you add such countries.

If you train a linear model on this data, you get the solid line, while the old model is represented by the dotted line. As you can see, not only does adding a few missing countries significantly alter the model, but it makes it clear that such a simple linear model is probably never going to work well. It seems that very rich countries are not happier than moderately rich countries (in fact, they seem slightly unhappier!), and conversely some poor countries seem happier than many rich countries.

By using a nonrepresentative training set, you trained a model that is unlikely to make accurate predictions, especially for very poor and very rich countries.

By using a nonrepresentative training set, you trained a model that is unlikely to make accurate predictions, especially for very poor and very rich countries.

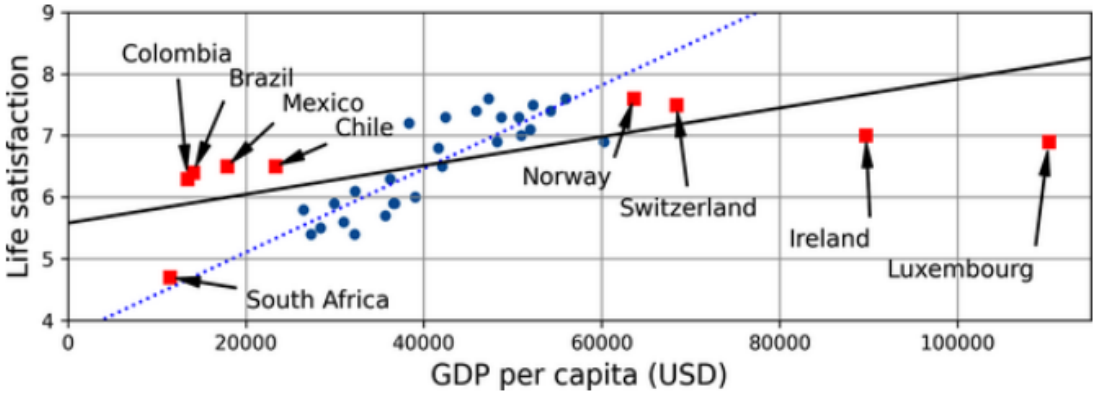


Figure 1-22. A more representative training sample

It is crucial to use a training set that is representative of the cases you want to generalize to. This is often harder than it sounds: if the sample is too small, you will have *sampling noise* (i.e., nonrepresentative data as a result of chance), but even very large samples can be nonrepresentative if the sampling method is flawed. This is called *sampling bias*.

Poor-Quality Data

Obviously, if your training data is full of errors, outliers, and noise (e.g., due to poor-quality measurements), it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well. It is often well worth the effort to spend time cleaning up your training data. The truth is, most data scientists spend a significant part of their time doing just that. The following are a couple examples of when you'd want to clean up training data:

- If some instances are clearly outliers, it may help to simply discard them or try to fix the errors manually.
- If some instances are missing a few features (e.g., 5% of your customers did not specify their age), you must decide whether you want to ignore this attribute altogether, ignore these instances, fill in the missing values (e.g., with the median age), or train one model with the feature and one model without it.

Irrelevant Features

As the saying goes: garbage in, garbage out. Your system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones. A critical part of the success of a machine learning project is coming up with a good set of features to train on. This process, called *feature engineering*, involves the following steps:

- *Feature selection* (selecting the most useful features to train on among existing features)
- *Feature extraction* (combining existing features to produce a more useful one—as we saw earlier, dimensionality reduction algorithms can help)
- Creating new features by gathering new data

Overfitting the Training Data

Say you are visiting a foreign country and the taxi driver rips you off. You might be tempted to say that *all* taxi drivers in that country are thieves. Overgeneralizing is something that we humans do all too often, and unfortunately machines can fall into the same trap if we are not careful. In machine learning this is called *overfitting*: it means that the model performs well on the training data, but it does not generalize well.

Figure 1-23 shows an example of a high-degree polynomial life satisfaction model that strongly overfits the training data. Even though it performs much better on the training data than the simple linear model, would you really trust its predictions?

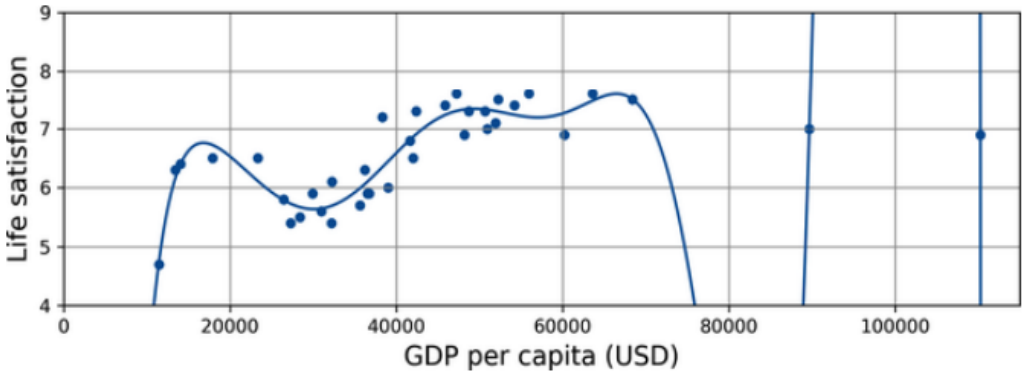


Figure 1-23. Overfitting the training data

WARNING

Overfitting happens when the model is too complex relative to the amount and noisiness of the training data. Here are possible solutions:

- Simplify the model by selecting one with fewer parameters (e.g., a linear model rather than a high-degree polynomial model), by reducing the number of attributes in the training data, or by constraining the model.
- Gather more training data.
- Reduce the noise in the training data (e.g., fix data errors and remove outliers).

Constraining a model to make it simpler and reduce the risk of overfitting is called *regularization*. For example, the linear model we defined earlier has two parameters, θ_0 and θ_1. This gives the learning algorithm two *degrees of freedom* to adapt the model to the training data: it can tweak both the height (θ_0) and the slope (θ_1) of the line. If we forced $\theta_1 = 0$, the algorithm would have only one degree of freedom and would have a much harder time fitting the data properly: all it could do is move the line up or down to get as close as possible to the training instances, so it would end up around the mean. A very simple model indeed! If we allow the algorithm to modify θ_1 but we force it to keep it small, then the learning algorithm will effectively have somewhere in between one and two degrees of freedom. It will produce a model that's simpler than one with two degrees of freedom, but more complex than one with just one. You want to find the right balance between fitting the training data perfectly and keeping the model simple enough to ensure that it will generalize well.

Underfitting the Training Data

As you might guess, *underfitting* is the opposite of overfitting: it occurs when your model is too simple to learn the underlying structure of the data. For example, a linear model of life satisfaction is prone to underfit; reality is just more complex than the model, so its predictions are bound to be inaccurate, even on the training examples.

Here are the main options for fixing this problem:

- Select a more powerful model, with more parameters.

- Feed better features to the learning algorithm (feature engineering).
- Reduce the constraints on the model (for example by reducing the regularization hyperparameter).

2(b) Explain the difference between random sampling and stratified

sampling. Write code-snippet to implement stratified sampling.

Random sampling involves randomly selecting data points from a dataset without considering any specific characteristics or attributes. On the other hand, stratified sampling involves dividing the dataset into subgroups based on important attributes and then randomly sampling from each subgroup to ensure that the sample is representative of the overall population.

To implement stratified sampling in Python using Scikit-Learn, you can use the `train_test_split` function with the `stratify` parameter. Here is an example code snippet:

```python

from sklearn.model_selection import train_test_split

import pandas as pd

# Assuming 'data' is your dataset and 'important_attribute' is the important attribute for stratified sampling

X = data.drop(columns=['target_column'])
```

y = data['target_column']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y)

```

In this code snippet, `train_test_split` is used to split the dataset into training and test sets while ensuring that the stratification is based on the 'target_column' (important attribute). This helps in creating a representative test set for more accurate evaluation of the model.

2(c) Identify how Grid Search and Randomized Search help in FineTuning a model. Write code-snippet as applicable.

Grid Search and Randomized Search help in fine-tuning a model by systematically searching through hyperparameter combinations to find the best set of values that optimize the model's performance. Grid Search exhaustively searches through all specified hyperparameter values, while Randomized Search evaluates a fixed number of combinations by randomly selecting values for each hyperparameter at every iteration.

Here is an example code snippet for Grid Search:

```python

from sklearn.model_selection import GridSearchCV

param_grid = [

  {'preprocessing__geo_n_clusters': [5, 8, 10],

   'random_forest__max_features': [2, 4, 6]}

]

grid_search = GridSearchCV(full_pipeline, param_grid, cv=3,
scoring='neg_root_mean_squared_error')

grid_search.fit(housing, housing_labels)

```

And for Randomized Search:

```python

from sklearn.model_selection import RandomizedSearchCV

```
from scipy.stats import randint

param_distribs = {
    'preprocessing__geo_n_clusters': randint(low=3, high=50),
    'random_forest__max_features': randint(low=2, high=20)
}

rnd_search = RandomizedSearchCV(full_pipeline, param_distributions=param_distribs, n_iter=16,
cv=3, scoring='neg_root_mean_squared_error', random_state=42)

rnd_search.fit(housing, housing_labels)
```

3(a) Apply Find-S algorithm to construct a maximally specific

hypothesis. Show its working over training instances in Table 1.

Find-S: Find Maximally Specific Hypothesis

- Initialize h to the most specific hypothesis in H.
- For each positive training instance x
  - For each attribute constraint a_i in h,
    If the constraint is satisfied by x_i
    Then do nothing
    Else replace a_i in h by the next more general constraint that is satisfied by x.
- Output hypothesis h.

Steps Involved In Find-S :

- Start with the most specific hypothesis.
  h = {φ, φ, φ, φ, φ, φ}
- Take the next example and if it is negative, then no changes occur to the hypothesis.
- If the example is positive and we find that our initial hypothesis is too specific then we update our current hypothesis to a general condition.
- Keep repeating the above steps till all the training examples are complete.
- After we have completed all the training examples we will have the final hypothesis when can use to classify the new examples.

Find-S: Example

Ex Num	Sky	Temp	Humid	Wind	Water	Forecst	EnjoySpt
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

Specific $\emptyset \rightarrow \{Sunny, Warm, etc.\} \rightarrow$ to General.

Example:

- Begin with $h = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$.
- Take example 1 from the table above: let's call this x.
- $h(x) = 0$, so replace each a_i in h with more general ones so that h is satisfied by x (i.e. $h(x) = 1$).

  $h \leftarrow \langle Sunny, Warm, Normal, Strong, Warm, Same \rangle$

Find-S: Example

Ex Num	Sky	Temp	Humid	Wind	Water	Forecst	EnjoySpt
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

Start with
$\langle Sunny, Warm, Normal, Strong, Warm, Same\rangle$:

- Take example 2 from the table above (call it x).
- $h(x) = 0$, so generalize conflicting constraints.

$$h \leftarrow \langle Sunny, Warm, ?, Strong, Warm, Same\rangle$$

- Take example 3: nothing happens, because it's a **negative example** (current h is **already consistent**: $h(x_3) = c(x_3)$)
- Take example 4: $h \leftarrow \langle Sunny, Warm, ?, Strong, ?, ?\rangle$

3(b) Apply List-Then-Eliminate algorithm to find set of hypotheses

consistent with the training instances shown in Table 1.

List-Then-Eliminate Algorithm

- Algorithm
  - $VS \leftarrow H$
  - For each $\langle x, c(x)\rangle \in D$
    remove from VS any hypothesis h for which $h(x) \neq c(x)$
  - Output the list of hypotheses in VS.
- Evaluation
  - Problem: Can't cope with infinite hypothesis space. Need to enumerate all.
  - Merit: guaranteed to output **all** consistent hypotheses, if H is finite.

LTE Illustration

Attribute instance:

- Sky: Sunny, Cloudy, Rainy
- AirTemp: Warm, Cold
- Humidity: Normal, High
- Wind: Strong, Weak
- Water: Warm, Cold
- Forecast: Same, Change

Initialize version space

#semantically distinct hypotheses :

1+4*3*3*3*3*3=973

Attribute instance:

- Sky: Sunny, Cloudy, Rainy
- AirTemp: Warm, Cold
- Humidity: Normal, High
- Wind: Strong, Weak
- Water: Warm, Cold
- Forecast: Same, Change

Initialize version space

#semantically distinct hypotheses :

1+4*3*3*3*3*3=973

x_1=<S,W,N,S,W,S>,+

<S,W,N,S,W,S>
<?,W,N,S,W,S>
<S,?,N,S,W,S>
<S,W,?,S,W,S>
...
<S,W,?,S,?,?>
...
<S,?,?,?,?,?>
<?,W,?,?,?,?>
<?,?,N,?,?,?>
<?,?,?,S,?,?>
<?,?,?,?,W,?>
<?,?,?,?,?,S>
...
<?, ?, ?, ?, ?, ?>

Attribute instance:

- Sky: Sunny, Cloudy, Rainy
- AirTemp: Warm, Cold
- Humidity: Normal, High
- Wind: Strong, Weak
- Water: Warm, Cold
- Forecast: Same, Change

x_1=<S,W,N,S,W,S>,+

x_2=<S,W,H,S,W,S>,+

Initialize version space

#semantically distinct hypotheses :

1+4*3*3*3*3*3=973

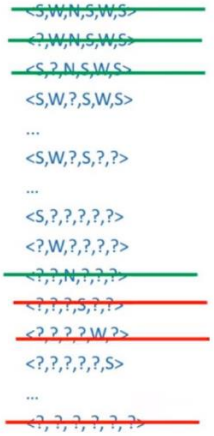
Attribute instance:

- Sky: Sunny, Cloudy, Rainy
- AirTemp: Warm, Cold
- Humidity: Normal, High
- Wind: Strong, Weak
- Water: Warm, Cold
- Forecast: Same, Change

x_1=<S,W,N,S,W,S>,+

x_2=<S,W,H,S,W,S>,+

x_3=<R,C,H,S,W,C>,-



Initialize version space

#semantically distinct hypotheses :

Attribute instance:

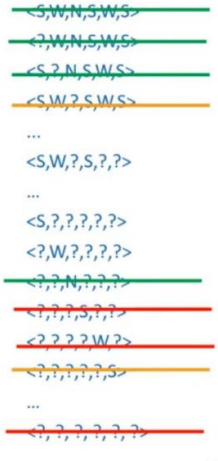
- Sky: Sunny, Cloudy, Rainy
- AirTemp: Warm, Cold
- Humidity: Normal, High
- Wind: Strong, Weak
- Water: Warm, Cold
- Forecast: Same, Change

x_1=<S,W,N,S,W,S>,+

x_2=<S,W,H,S,W,S>,+

x_3=<R,C,H,S,W,C>,-

x_4=<S,W,H,S,C,C>,+



Initialize version space

#semantically distinct hypotheses :

3(c)

Outline the concepts involved in the following. Write codesnippet as applicable.

i. Measuring accuracy using Cross-Validation

ii. Confusion Matrix

iii. Precision and Recall

Measuring Accuracy Using Cross-Validation

A good way to evaluate a model is to use cross-validation, just as you did in Chapter 2. Let's use the `cross_val_score()` function to evaluate our

`SGDClassifier` model, using *k*-fold cross-validation with three folds. Remember that *k*-fold cross-validation means splitting the training set into *k* folds (in this case, three), then training the model *k* times, holding out a different fold each time for evaluation (see Chapter 2):

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.95035, 0.96035, 0.9604 ])
```

Wow! Above 95% accuracy (ratio of correct predictions) on all cross-validation folds? This looks amazing, doesn't it? Well, before you get too excited, let's look at a dummy classifier that just classifies every single image in the most frequent class, which in this case is the negative class (i.e., *non* 5):

```
from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier()
dummy_clf.fit(X_train, y_train_5)
print(any(dummy_clf.predict(X_train)))  # prints False: no 5s detected
```

Can you guess this model's accuracy? Let's find out:

```
>>> cross_val_score(dummy_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.90965, 0.90965, 0.90965])
```

That's right, it has over 90% accuracy! This is simply because only about 10% of the images are 5s, so if you always guess that an image is *not* a 5, you will be right about 90% of the time. Beats Nostradamus.

This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with *skewed datasets* (i.e., when some classes are much more frequent than others). A much better way to evaluate the performance of a classifier is to look at the *confusion matrix* (CM).

Confusion Matrices

The general idea of a confusion matrix is to count the number of times instances of class A are classified as class B, for all A/B pairs. For example, to know the number of times the classifier confused images of 8s with 0s, you would look at row #8, column #0 of the confusion matrix.

To compute the confusion matrix, you first need to have a set of predictions so that they can be compared to the actual targets. You could make predictions on the test set, but it's best to keep that untouched for now (remember that you want to use the test set only at the very end of your project, once you have a classifier that you are ready to launch). Instead, you can use the `cross_val_predict()` function:

```
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

Just like the `cross_val_score()` function, `cross_val_predict()` performs *k*-fold cross-validation, but instead of returning the evaluation scores, it returns the predictions made on each test fold. This means that you get a clean prediction for each instance in the training set (by "clean" I mean "out-of-sample": the model makes predictions on data that it never saw during training).

Now you are ready to get the confusion matrix using the `confusion_matrix()` function. Just pass it the target classes (`y_train_5`) and the predicted classes (`y_train_pred`):

```
>>> from sklearn.metrics import confusion_matrix
>>> cm = confusion_matrix(y_train_5, y_train_pred)
>>> cm
array([[53892,   687],
       [ 1891,  3530]])
```

Each row in a confusion matrix represents an *actual class*, while each column represents a *predicted class*. The first row of this matrix considers non-5 images (the *negative class*): 53,892 of them were correctly classified as non-5s (they are called *true negatives*), while the remaining 687 were wrongly classified as 5s (*false positives*, also called *type I errors*). The second row considers the images of 5s (the *positive class*): 1,891 were wrongly classified as non-5s (*false negatives*, also called *type II errors*), while the remaining 3,530 were correctly classified as 5s (*true positives*). A perfect classifier would only have true positives and true negatives, so its

confusion matrix would have nonzero values only on its main diagonal (top left to bottom right):

```
>>> y_train_perfect_predictions = y_train_5  # pretend we reached perfection
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579,     0],
       [    0,  5421]])
```

The confusion matrix gives you a lot of information, but sometimes you may prefer a more concise metric. An interesting one to look at is the accuracy of the positive predictions; this is called the *precision* of the classifier (Equation 3-1).

Equation 3-1. Precision

$$\text{precision} = \frac{TP}{TP + FP}$$

TP is the number of true positives, and *FP* is the number of false positives.

A trivial way to have perfect precision is to create a classifier that always makes negative predictions, except for one single positive prediction on the instance it's most confident about. If this one prediction is correct, then the classifier has 100% precision (precision = 1/1 = 100%). Obviously, such a classifier would not be very useful, since it would ignore all but one positive instance. So, precision is typically used along with another metric named *recall*, also called *sensitivity* or the *true positive rate* (TPR): this is the ratio of positive instances that are correctly detected by the classifier (Equation 3-2).

Equation 3-2. Recall

$$\text{recall} = \frac{TP}{TP + FN}$$

FN is, of course, the number of false negatives.

If you are confused about the confusion matrix, Figure 3-3 may help.

Precision and Recall

Scikit-Learn provides several functions to compute classifier metrics, including precision and recall:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred)  # == 3530 / (687 + 3530)
0.8370879772350012
>>> recall_score(y_train_5, y_train_pred)  # == 3530 / (1891 + 3530)
0.6511713705958311
```

Now our 5-detector does not look as shiny as it did when we looked at its accuracy. When it claims an image represents a 5, it is correct only 83.7% of the time. Moreover, it only detects 65.1% of the 5s.

It is often convenient to combine precision and recall into a single metric called the *F_1 score*, especially when you need a single metric to compare two classifiers. The F_1 score is the *harmonic mean* of precision and recall (Equation 3-3). Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values. As a result, the classifier will only get a high F_1 score if both recall and precision are high.

Equation 3-3. F_1 score

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

To compute the F_1 score, simply call the `f1_score()` function:

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7325171197343846
```

The F_1 score favors classifiers that have similar precision and recall. This is not always what you want: in some contexts you mostly care about precision, and in other contexts you really care about recall. For example, if you trained a classifier to detect videos that are safe for kids, you would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a much higher recall but lets a few really bad videos show up in your product (in such cases, you may even want to add a human pipeline to check the classifier's video selection). On the other hand, suppose you train a classifier to detect shoplifters in surveillance images: it is probably fine if your classifier only has 30% precision as long as it has 99% recall (sure, the security guards will get a few false alerts, but almost all shoplifters will get caught).

Unfortunately, you can't have it both ways: increasing precision reduces recall, and vice versa. This is called the *precision/recall trade-off*.

4(a)

Identify how the number of semantically distinct hypotheses in

the hypotheses space is only 973 considering the training

instances in Table 1. Also explain how this number will increase

with the inclusion of additional categorical attributes in the training instances.

4(b)

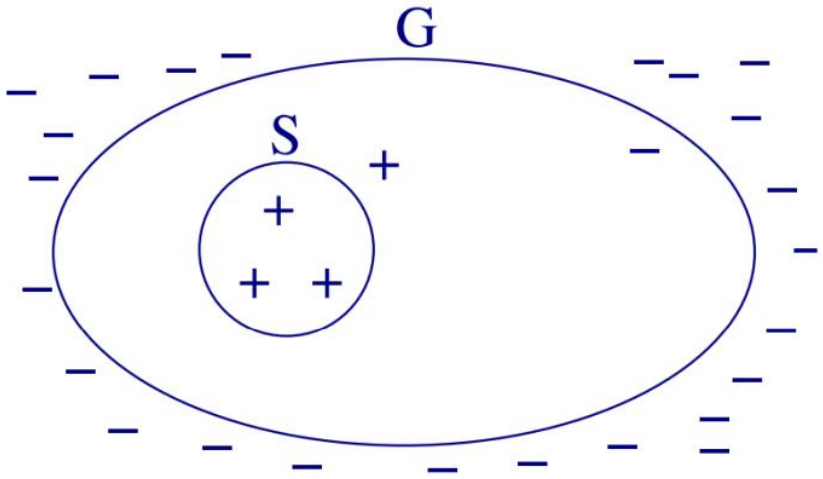
Apply Candidate

-Elimination algorithm to construct boundary of

most specific and most general hypothesis set by referring

training instances in Table 1.

Candidate Elimination Algorithm: Basic Concept



- If negative example is misclassified by hypotheses in G, reduce the scope (make more specific).
- If positive example is misclassified by hypotheses in S, increase the scope (make more general).

Candidate Elimination Algorithm

$G \leftarrow$ maximally general hypotheses in H: $\langle ?, ?, ?, ?, ?, ? \rangle$
$S \leftarrow$ maximally specific hypotheses in H: $\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$

For each training example d, do

- If d is a positive example
  - Remove from G any hypothesis inconsistent with d
  - For each hypothesis s in S that is not consistent with d
    - Remove s from S
    - Add to S all minimal generalizations h of s such that
      1. h is consistent with d, and
      2. some member of G is more general than h
    - Remove from S any hypothesis that is more general than another hypothesis in S
- If d is a negative example
  - Remove from S any hypothesis inconsistent with d
  - For each hypothesis g in G that is not consistent with d
    - Remove g from G
    - Add to G all minimal specializations h of g such that
      1. h is consistent with d, and
      2. some member of S is more specific than h
    - Remove from G any hypothesis that is less general than another hypothesis in G

- Step 1: Load the Data set.
- Step 2: Initialize General Hypothesis ‘G’ and Specific Hypothesis ‘S’.
- Step 3: For each training example,
  if example is positive example:
    Make specific hypothesis more general.
    if attribute_value == hypothesis_value:
      Do nothing
    else:
      replace attribute value with ‘?’ (Basically generalizing it)
- Step 4: If example is Negative example:
  Make generalize hypothesis more specific.

Example

Ex Num	Sky	Temp	Humid	Wind	Water	Forecst	EnjoySpt
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

Initial $S_0 = \{\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\}$, $G_0 = \{\langle ?, ?, ?, ?, ?, ? \rangle\}$.

- After example 1:

$$S_1 = \{\langle Sunny, Warm, Normal, Strong, Warm, Same \rangle\}$$

- After example 2:

$$S_2 = \{\langle Sunny, Warm, ?, Strong, Warm, Same \rangle\}$$

Note that $G_0 = G_1 = G_2$, i.e., no change.

Ex Num	Sky	Temp	Humid	Wind	Water	Forecst	EnjoySpt
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

$S_2 = \{\langle Sunny, Warm, ?, Strong, Warm, Same\rangle\}$, and
$G_2 = \{\langle ?, ?, ?, ?, ?, ?\rangle\}$.

- After example 3, a negative example, G gets updated:

$$G_3 = \{\langle Sunny, ?, ?, ?, ?, ?\rangle, \langle ?, Warm, ?, ?, ?, ?\rangle \langle ?, ?, ?, ?, ?, Same\rangle\}$$

- Question: Why didn't $\langle ?, ?, Normal, ?, ?, ?\rangle$ etc. get added to G?: See the **"more specific than"** condition.

Ex Num	Sky	Temp	Humid	Wind	Water	Forecst	EnjoySpt
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

$S_3 = \{\langle Sunny, Warm, ?, Strong, Warm, Same\rangle\}$, and
$G_3 = \{\langle Sunny, ?, ?, ?, ?, ?\rangle, \langle ?, Warm, ?, ?, ?, ?\rangle, \langle ?, ?, ?, ?, ?, Same\rangle\}$.

- After example 4:

$$S_4 = \{\langle Sunny, Warm, ?, Strong, ?, ?\rangle\}$$

$$G_4 = \{\langle Sunny, ?, ?, ?, ?, ?\rangle, \langle ?, Warm, ?, ?, ?, ?\rangle\}$$

- Both S and G got updated.

Initially : G = [[?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, ?],
[?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, ?]]
S = [Null, Null, Null, Null, Null, Null]

For instance 1 : <'sunny','warm','normal','strong','warm ','same'> and positive output.
G1 = G
S1 = ['sunny','warm','normal','strong','warm ','same']

For instance 2 : <'sunny','warm','high','strong','warm ','same'> and positive output.
G2 = G
S2 = ['sunny','warm',?,'strong','warm ','same']

For instance 3 : <'rainy','cold','high','strong','warm ','change'> and negative output.
G3 = [['sunny', ?, ?, ?, ?, ?], [?, 'warm', ?, ?, ?, ?], [?, ?, ?, ?, ?, ?],
[?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, ?], [?, ?, ?, ?, ?, 'same']]
S3 = S2

For instance 4 : <'sunny','warm','high','strong','cool','change'> and positive output.
G4 = G3
S4 = ['sunny','warm',?,'strong', ?, ?]

At last, by synchronizing the G4 and S4 algorithm produce the output.

Output :

G = [['sunny', ?, ?, ?, ?, ?], [?, 'warm', ?, ?, ?, ?]]
S = ['sunny','warm',?,'strong', ?, ?]

4(c) Outline the concept of Precision/Recall trade

-off with an

example.

The Precision/Recall Trade-off

To understand this trade-off, let's look at how the `SGDClassifier` makes its classification decisions. For each instance, it computes a score based on a *decision function*. If that score is greater than a threshold, it assigns the instance to the positive class; otherwise it assigns it to the negative class. Figure 3-4 shows a few digits positioned from the lowest score on the left to the highest score on the right. Suppose the *decision threshold* is positioned at the central arrow (between the two 5s): you will find 4 true positives (actual

5s) on the right of that threshold, and 1 false positive (actually a 6). Therefore, with that threshold, the precision is 80% (4 out of 5). But out of 6 actual 5s, the classifier only detects 4, so the recall is 67% (4 out of 6). If you raise the threshold (move it to the arrow on the right), the false positive (the 6) becomes a true negative, thereby increasing the precision (up to 100% in this case), but one true positive becomes a false negative, decreasing recall down to 50%. Conversely, lowering the threshold increases recall and reduces precision.

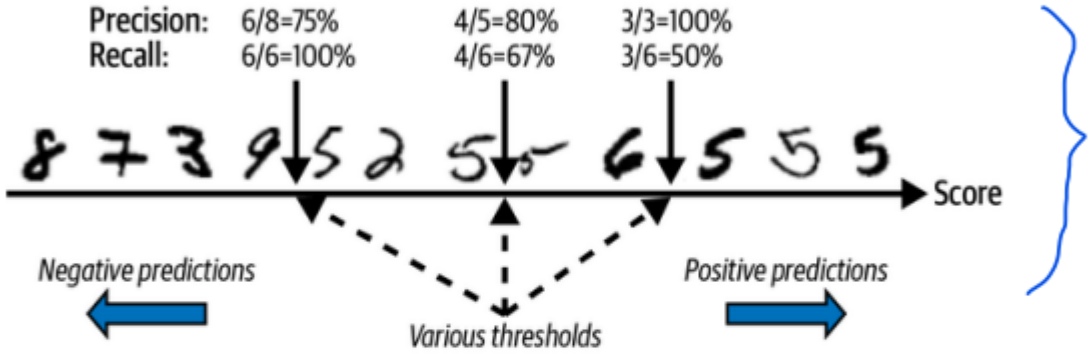


Figure 3-4. The precision/recall trade-off: images are ranked by their classifier score, and those above the chosen decision threshold are considered positive; the higher the threshold, the lower the recall, but (in general) the higher the precision

Scikit-Learn does not let you set the threshold directly, but it does give you access to the decision scores that it uses to make predictions. Instead of calling the classifier's `predict()` method, you can call its `decision_function()` method, which returns a score for each instance, and then use any threshold you want to make predictions based on those scores:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([2164.22030239])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True])
```

The `SGDClassifier` uses a threshold equal to 0, so the preceding code returns the same result as the `predict()` method (i.e., `True`). Let's raise the threshold:

```
>>> threshold = 3000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False])
```

This confirms that raising the threshold decreases recall. The image actually represents a 5, and the classifier detects it when the threshold is 0, but it misses it when the threshold is increased to 3,000.

How do you decide which threshold to use? First, use the `cross_val_predict()` function to get the scores of all instances in the training set, but this time specify that you want to return decision scores instead of predictions:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

With these scores, use the `precision_recall_curve()` function to compute precision and recall for all possible thresholds (the function adds a last precision of 0 and a last recall of 1, corresponding to an infinite threshold):

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

Finally, use Matplotlib to plot precision and recall as functions of the threshold value (Figure 3-5). Let's show the threshold of 3,000 we selected:

```
plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
plt.vlines(threshold, 0, 1.0, "k", "dotted", label="threshold")
[...]  # beautify the figure: add grid, legend, axis, labels, and circles
plt.show()
```

CHAPTER: Machine learning Landscape

1. Explain Machine Learning with examples of applications.

What Is Machine Learning?

Machine learning is the science (and art) of programming computers so they can *learn from data*.

Here is a slightly more general definition:

> *[Machine learning is the] field of study that gives computers the ability to learn without being explicitly programmed.*
>
> —Arthur Samuel, 1959

And a more engineering-oriented one:

> *A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.*
>
> —Tom Mitchell, 1997

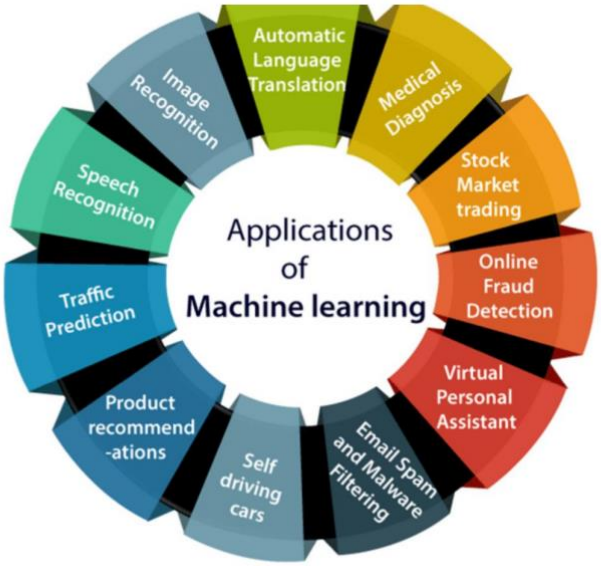
Your spam filter is a machine learning program that, given examples of spam emails (flagged by users) and examples of regular emails (nonspam, also called "ham"), can learn to flag spam. The examples that the system uses to learn are called the *training set*. Each training example is called a *training instance* (or *sample*). The part of a machine learning system that learns and makes predictions is called a *model*. Neural networks and random forests are examples of models.

In this case, the task *T* is to flag spam for new emails, the experience *E* is the *training data*, and the performance measure *P* needs to be defined; for example, you can use the ratio of correctly classified emails. This particular performance measure is called *accuracy*, and it is often used in classification tasks.

There are separate metrics for regression tasks.

If you just download a copy of all Wikipedia articles, your computer has a lot more data, but it is not suddenly better at any task. This is not machine learning.

Applications of Machine learning



Examples of Applications

Let's look at some concrete examples of machine learning tasks, along with the techniques that can tackle them:

Analyzing images of products on a production line to automatically classify them

This is image classification, typically performed using convolutional neural networks (CNNs; see Chapter 14) or sometimes transformers (see Chapter 16).

Detecting tumors in brain scans

This is semantic image segmentation, where each pixel in the image is classified (as we want to determine the exact location and shape of tumors), typically using CNNs or transformers.

Automatically classifying news articles

This is natural language processing (NLP), and more specifically text classification, which can be tackled using recurrent neural networks (RNNs) and CNNs, but transformers work even better (see Chapter 16).

Automatically flagging offensive comments on discussion forums

This is also text classification, using the same NLP tools.

Summarizing long documents automatically

This is a branch of NLP called text summarization, again using the same tools.

Creating a chatbot or a personal assistant

This involves many NLP components, including natural language understanding (NLU) and question-answering modules.

Forecasting your company's revenue next year, based on many performance metrics

This is a regression task (i.e., predicting values) that may be tackled using any regression model, such as a linear regression or polynomial regression model (see Chapter 4), a regression support vector machine (see Chapter 5), a regression random forest (see Chapter 7), or an artificial neural network (see Chapter 10). If you want to take into account sequences of past performance metrics, you may want to use RNNs, CNNs, or transformers (see Chapters 15 and 16).

Making your app react to voice commands

This is speech recognition, which requires processing audio samples: since they are long and complex sequences, they are typically processed using RNNs, CNNs, or transformers (see Chapters 15 and 16).

Detecting credit card fraud

This is anomaly detection, which can be tackled using isolation forests, Gaussian mixture models (see Chapter 9), or autoencoders (see Chapter 17).

Segmenting clients based on their purchases so that you can design a different marketing strategy for each segment

This is clustering, which can be achieved using *k*-means, DBSCAN, and more (see Chapter 9).

Representing a complex, high-dimensional dataset in a clear and insightful diagram

This is data visualization, often involving dimensionality reduction techniques (see Chapter 8).

Recommending a product that a client may be interested in, based on past purchases

This is a recommender system. One approach is to feed past purchases (and other information about the client) to an artificial neural network (see Chapter 10), and get it to output the most likely next purchase. This

neural net would typically be trained on past sequences of purchases across all clients.

Building an intelligent bot for a game

This is often tackled using reinforcement learning (RL; see Chapter 18), which is a branch of machine learning that trains agents (such as bots) to pick the actions that will maximize their rewards over time (e.g., a bot may get a reward every time the player loses some life points), within a given environment (such as the game). The famous AlphaGo program that beat the world champion at the game of Go was built using RL.

2. Explain Supervised Learning and Unsupervised Learning with examples.(repeated)

3. Discuss any four main challenges of Machine Learning. .(repeated)

4. Discuss the differences between Online and Batch Learning systems.

Batch Versus Online Learning

Another criterion used to classify machine learning systems is whether or not the system can learn incrementally from a stream of incoming data.

Batch learning

In *batch learning*, the system is incapable of learning incrementally: it must be trained using all the available data. This will generally take a lot of time and computing resources, so it is typically done offline. First the system is trained, and then it is launched into production and runs without learning anymore; it just applies what it has learned. This is called *offline learning*.

Unfortunately, a model's performance tends to decay slowly over time, simply because the world continues to evolve while the model remains unchanged. This phenomenon is often called *model rot* or *data drift*. The solution is to regularly retrain the model on up-to-date data. How often you need to do that depends on the use case: if the model classifies pictures of cats and dogs, its performance will decay very slowly, but if the model deals with fast-evolving systems, for example making predictions on the financial market, then it is likely to decay quite fast.

If you want a batch learning system to know about new data (such as a new type of spam), you need to train a new version of the system from scratch on the full dataset (not just the new data, but also the old data), then replace the old model with the new one. Fortunately, the whole process of training, evaluating, and launching a machine learning system can be automated fairly easily (as we saw in Figure 1-3), so even a batch learning system can adapt

---

to change. Simply update the data and train a new version of the system from scratch as often as needed.

This solution is simple and often works fine, but training using the full set of data can take many hours, so you would typically train a new system only every 24 hours or even just weekly. If your system needs to adapt to rapidly changing data (e.g., to predict stock prices), then you need a more reactive solution.

Also, training on the full set of data requires a lot of computing resources (CPU, memory space, disk space, disk I/O, network I/O, etc.). If you have a lot of data and you automate your system to train from scratch every day, it will end up costing you a lot of money. If the amount of data is huge, it may even be impossible to use a batch learning algorithm.

Finally, if your system needs to be able to learn autonomously and it has limited resources (e.g., a smartphone application or a rover on Mars), then carrying around large amounts of training data and taking up a lot of resources to train for hours every day is a showstopper.

A better option in all these cases is to use algorithms that are capable of learning incrementally.

Online learning

In *online learning*, you train the system incrementally by feeding it data instances sequentially, either individually or in small groups called *mini-batches*. Each learning step is fast and cheap, so the system can learn about new data on the fly, as it arrives (see Figure 1-14).

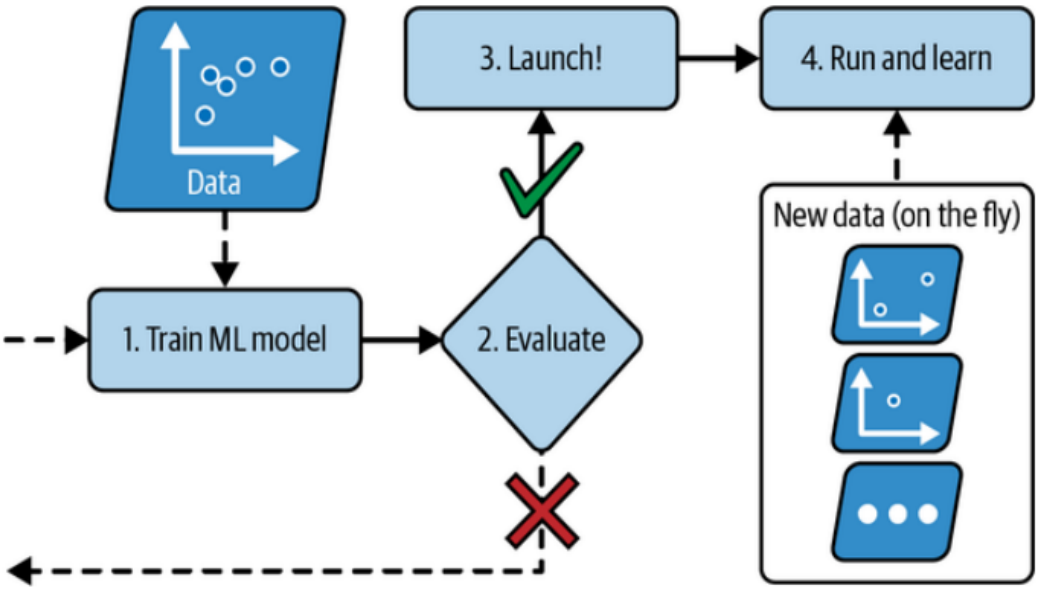


Figure 1-14. In online learning, a model is trained and launched into production, and then it keeps learning as new data comes in

Online learning is useful for systems that need to adapt to change extremely rapidly (e.g., to detect new patterns in the stock market). It is also a good option if you have limited computing resources; for example, if the model is trained on a mobile device.

Additionally, online learning algorithms can be used to train models on huge datasets that cannot fit in one machine's main memory (this is called *out-of-core* learning). The algorithm loads part of the data, runs a training step on that data, and repeats the process until it has run on all of the data (see Figure 1-15).

A big challenge with online learning is that if bad data is fed to the system, the system's performance will decline, possibly quickly (depending on the

data quality and learning rate). If it's a live system, your clients will notice. For example, bad data could come from a bug (e.g., a malfunctioning sensor on a robot), or it could come from someone trying to game the system (e.g., spamming a search engine to try to rank high in search results). To reduce this risk, you need to monitor your system closely and promptly switch learning off (and possibly revert to a previously working state) if you detect a drop in performance. You may also want to monitor the input data and react to abnormal data; for example, using an anomaly detection algorithm (see Chapter 9).

5. Explain Overfitting and Underfitting training data with appropriate figures.(repeated)

6. Contrast Instance-based versus Model-based learning systems.

Instance-Based Versus Model-Based Learning

One more way to categorize machine learning systems is by how they *generalize*. Most machine learning tasks are about making predictions. This means that given a number of training examples, the system needs to be able to make good predictions for (generalize to) examples it has never seen before. Having a good performance measure on the training data is good, but insufficient; the true goal is to perform well on new instances.

There are two main approaches to generalization: instance-based learning and model-based learning.

Instance-based learning

Possibly the most trivial form of learning is simply to learn by heart. If you were to create a spam filter this way, it would just flag all emails that are identical to emails that have already been flagged by users—not the worst solution, but certainly not the best.

Instead of just flagging emails that are identical to known spam emails, your spam filter could be programmed to also flag emails that are very similar to known spam emails. This requires a *measure of similarity* between two emails. A (very basic) similarity measure between two emails could be to count the number of words they have in common. The system would flag an email as spam if it has many words in common with a known spam email.

This is called *instance-based learning*: the system learns the examples by heart, then generalizes to new cases by using a similarity measure to compare them to the learned examples (or a subset of them). For example, in Figure 1-16 the new instance would be classified as a triangle because the majority of the most similar instances belong to that class.

Figure 1-16. Instance-based learning

Model-based learning and a typical machine learning workflow

Another way to generalize from a set of examples is to build a model of these examples and then use that model to make *predictions*. This is called *model-based learning* (Figure 1-17).

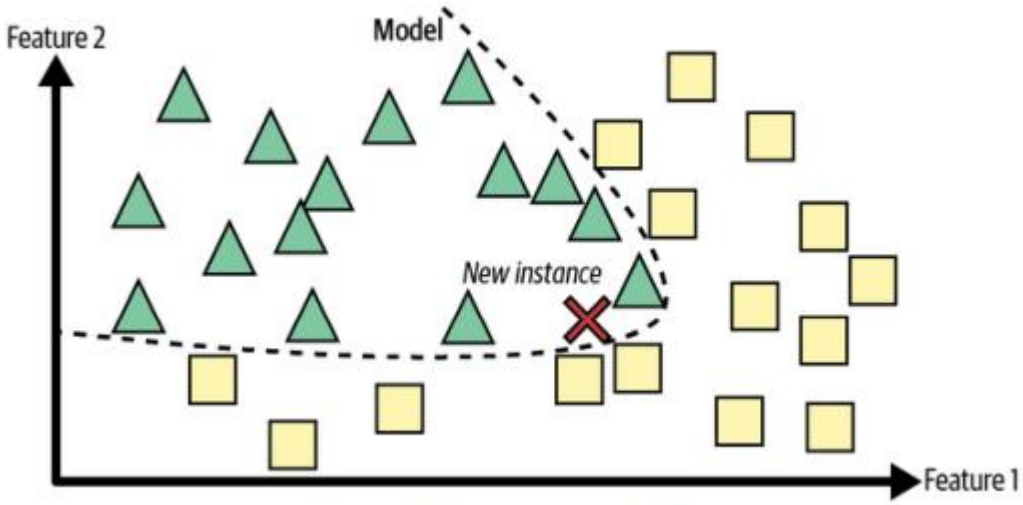


Figure 1-17. Model-based learning

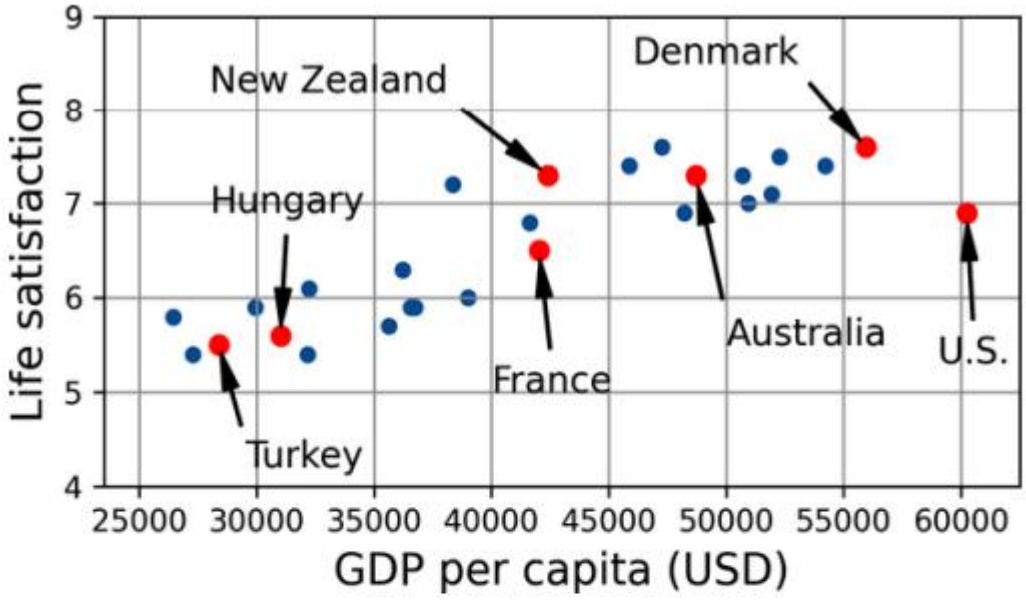


Figure 1-18. Do you see a trend here?

There does seem to be a trend here! Although the data is *noisy* (i.e., partly random), it looks like life satisfaction goes up more or less linearly as the country's GDP per capita increases. So you decide to model life satisfaction as a linear function of GDP per capita. This step is called *model selection*: you selected a *linear model* of life satisfaction with just one attribute, GDP per capita (Equation 1-1).

Equation 1-1. A simple linear model

$$\text{life\_satisfaction} = \theta_0 + \theta_1 \times \text{GDP\_per\_capita}$$

This model has two *model parameters*, θ_0 and θ_1.[4] By tweaking these parameters, you can make your model represent any linear function, as shown in Figure 1-19.

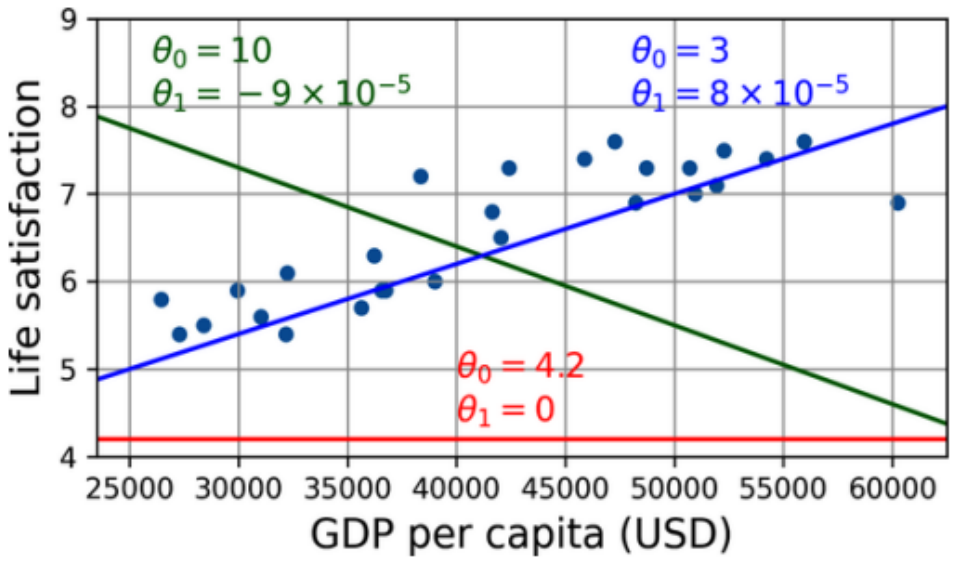


Figure 1-19. A few possible linear models

Before you can use your model, you need to define the parameter values θ_0 and θ_1. How can you know which values will make your model perform best? To answer this question, you need to specify a performance measure. You can either define a *utility function* (or *fitness function*) that measures how *good* your model is, or you can define a *cost function* that measures how *bad* it is. For linear regression problems, people typically use a cost function that measures the distance between the linear model's predictions and the training examples; the objective is to minimize this distance.

This is where the linear regression algorithm comes in: you feed it your training examples, and it finds the parameters that make the linear model fit best to your data. This is called *training* the model. In our case, the algorithm finds that the optimal parameter values are $\theta_0 = 3.75$ and $\theta_1 = 6.78 \times 10^{-5}$.

Model-based approach

CHAPTER: End-to-end Machine Learning Project

1. Explain with expressions a few of the popular metrics used in regression tasks.

2. Explain the difference between random sampling and stratified sampling and write code-snippet to show usage of the latter one.[repeated]

3. In context to prepare the data for Machine Learning algorithms, write a note on

a. Data Cleaning

b. Handling text and categorical attributes

c. Feature scaling[repeated]

4. Explain why model evaluation using k-Fold Cross Validation is considered a better

alternative over standard one set of train and validation set.[repeated]

5. With the code snippets, show how Grid Search and Randomized Search help in Fine-Tuning a model.[repeated]

MODULE 2

CHAPTER: Concept Learning and Learning Problems

1. Apply Find-S algorithm to construct a maximally specific hypothesis and show its working by taking the EnjoySport concept and training instances given in Table 1.[repeated]

2. Identify how the number of semantically distinct hypotheses in the hypotheses space is only 973 considering the training instances in Table 1. Also explain how this number will increase with the inclusion of additional categorical attributes in the training instances. [repeated]

3. Apply List-Then-Eliminate algorithm to find set of hypotheses consistent with the training example shown in Table 1. [repeated]

4. Apply Candidate-Elimination algorithm to construct boundary of most specific and general hypothesis set by referring training instances in Table 1. [repeated]

CHAPTER: Classification

1. Using code snippets, outline the concepts involved in

a. Measuring accuracy using Cross-Validation

b. Confusion Matrix

c. Precision and Recall

d. F1 Score [repeated]

2. Outline the concept of Precision/Recall trade-off with an example.

3. With the code snippet explain how Multilabel classification different from multiclass Multioutput classification?

Multilabel Classification

Until now, each instance has always been assigned to just one class. But in some cases you may want your classifier to output multiple classes for each instance. Consider a face-recognition classifier: what should it do if it recognizes several people in the same picture? It should attach one tag per person it recognizes. Say the classifier has been trained to recognize three faces: Alice, Bob, and Charlie. Then when the classifier is shown a picture of Alice and Charlie, it should output `[True, False, True]` (meaning "Alice yes, Bob no, Charlie yes"). Such a classification system that outputs multiple binary tags is called a *multilabel classification* system.

We won't go into face recognition just yet, but let's look at a simpler example, just for illustration purposes:

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= '7')
y_train_odd = (y_train.astype('int8') % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

This code creates a `y_multilabel` array containing two target labels for each digit image: the first indicates whether or not the digit is large (7, 8, or 9), and the second indicates whether or not it is odd. Then the code creates a `KNeighborsClassifier` instance, which supports multilabel classification (not all classifiers do), and trains this model using the multiple targets array. Now you can make a prediction, and notice that it outputs two labels:

```
>>> knn_clf.predict([some_digit])
array([[False,  True]])
```

And it gets it right! The digit 5 is indeed not large (`False`) and odd (`True`).

There are many ways to evaluate a multilabel classifier, and selecting the right metric really depends on your project. One approach is to measure the F_1 score for each individual label (or any other binary classifier metric discussed earlier), then simply compute the average score. The following code computes the average F_1 score across all labels:

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel, cv=3)
>>> f1_score(y_multilabel, y_train_knn_pred, average="macro")
0.976410265560605
```

This approach assumes that all labels are equally important, which may not be the case. In particular, if you have many more pictures of Alice than of Bob or Charlie, you may want to give more weight to the classifier's score on pictures of Alice. One simple option is to give each label a weight equal to its *support* (i.e., the number of instances with that target label). To do this, simply set `average="weighted"` when calling the `f1_score()` function.[5]

If you wish to use a classifier that does not natively support multilabel classification, such as `SVC`, one possible strategy is to train one model per label. However, this strategy may have a hard time capturing the dependencies between the labels. For example, a large digit (7, 8, or 9) is twice more likely to be odd than even, but the classifier for the "odd" label does not know what the classifier for the "large" label predicted. To solve this issue, the models can be organized in a chain: when a model makes a prediction, it uses the input features plus all the predictions of the models that come before it in the chain.

The good news is that Scikit-Learn has a class called `ChainClassifier` that does just that! By default it will use the true labels for training, feeding each model the appropriate labels depending on their position in the chain. But if you set the `cv` hyperparameter, it will use cross-validation to get "clean" (out-of-sample) predictions from each trained model for every instance in the training set, and these predictions will then be used to train all the models later in the chain. Here's an example showing how to create and train a `ChainClassifier` using the cross-validation strategy. As earlier, we'll just use the first 2,000 images in the training set to speed things up:

```
from sklearn.multioutput import ClassifierChain

chain_clf = ClassifierChain(SVC(), cv=3, random_state=42)
chain_clf.fit(X_train[:2000], y_multilabel[:2000])
```

Now we can use this `ChainClassifier` to make predictions:

```
>>> chain_clf.predict([some_digit])
array([[0., 1.]])
```

Multioutput Classification

The last type of classification task we'll discuss here is called *multioutput–multiclass classification* (or just *multioutput classification*). It is a generalization of multilabel classification where each label can be multiclass (i.e., it can have more than two possible values).

To illustrate this, let's build a system that removes noise from images. It will take as input a noisy digit image, and it will (hopefully) output a clean digit image, represented as an array of pixel intensities, just like the MNIST images. Notice that the classifier's output is multilabel (one label per pixel) and each label can have multiple values (pixel intensity ranges from 0 to 255). This is thus an example of a multioutput classification system.

> **NOTE**
>
> The line between classification and regression is sometimes blurry, such as in this example. Arguably, predicting pixel intensity is more akin to regression than to classification. Moreover, multioutput systems are not limited to classification tasks; you could even have a system that outputs multiple labels per instance, including both class labels and value labels.

Let's start by creating the training and test sets by taking the MNIST images and adding noise to their pixel intensities with NumPy's `randint()` function. The target images will be the original images:

```
np.random.seed(42)  # to make this code example reproducible
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

Let's take a peek at the first image from the test set (Figure 3-12). Yes, we're snooping on the test data, so you should be frowning right now.