

Experiment 4 - Supervised Learning

In this experiment, we will explore supervised learning techniques for regression and classification tasks. We will use Python and the Scikit-learn library to implement linear regression, polynomial regression, random forest classifier, and SVM models. We will also evaluate the models using appropriate evaluation measures.

Scikit-learn installation

To install the Scikit-learn library, you can use the following command

```
$ pip install -U scikit-learn
```

In order to check your installation you can use

```
$ python -m pip show scikit-learn # to see which version and where
      scikit-learn is installed
$ python -m pip freeze # to see all packages installed in the active
      virtualenv
$ python -c "import sklearn; sklearn.show_versions()"
```

Note that in order to avoid potential conflicts with other packages it is strongly recommended to use a virtual environment (venv) or a conda environment.

1.1 Regression

Regression is a statistical technique that relates a continuous dependent variable to one or more independent variables. In this part, we will fit different regression models on a toy dataset.

1.1.1 Generating examples

We will start by generating a toy dataset. Generate 40 examples using the following function

$$f(x) = \sin(x) + \epsilon,$$

where ϵ is sampled from a normal distribution with standard deviation 0.1, and $x \in [0, 1]$. Split the samples to into two equal sets: training set, and testing set.

Code Snippet 1.1: Generating examples.

```
import numpy as np

def true_fun(X):
    return np.sin(1.5 * np.pi * X)

np.random.seed(0)
n_samples = 40

X = np.random.rand(n_samples)
X_train = np.sort(X[:n_samples//2])
y_train = true_fun(X_train) + np.random.randn(n_samples//2) * 0.1

X_test = np.sort(X[n_samples//2:])
y_test = true_fun(X_test) + np.random.randn(n_samples//2) * 0.1
```

Plot both the training and testing examples in a scatter plot and show the true function curve for the range [0, 1).

Code Snippet 1.2: Plotting the dataset.

```
import matplotlib.pyplot as plt

x = np.linspace(0, 1, 100)
plt.plot(x, true_fun(x), label="True function")
plt.scatter(X_train, y_train, edgecolor="b", s=20, label="Train Examples")
plt.scatter(X_test, y_test, edgecolor="r", s=20, label="Test Examples")
plt.xlabel("x")
plt.ylabel("y")
plt.xlim((0, 1))
plt.ylim((-1.5, 1.5))

plt.legend(loc="best")
plt.title("Toy Dataset")
plt.show()
```

You should get a figure similar to Figure 1.1

1.1.2 Linear regression

Let's begin with a simple linear regression model. I.e., we will fit a line to the training set of the form

$$h(x) = w_0 + w_1 x,$$

where x is the input feature, w_0 is the intercept of the fitted line, and w_1 is its slope. We will use the linear regression implementation from *sklearn* to get our model. Plot the fitted line.

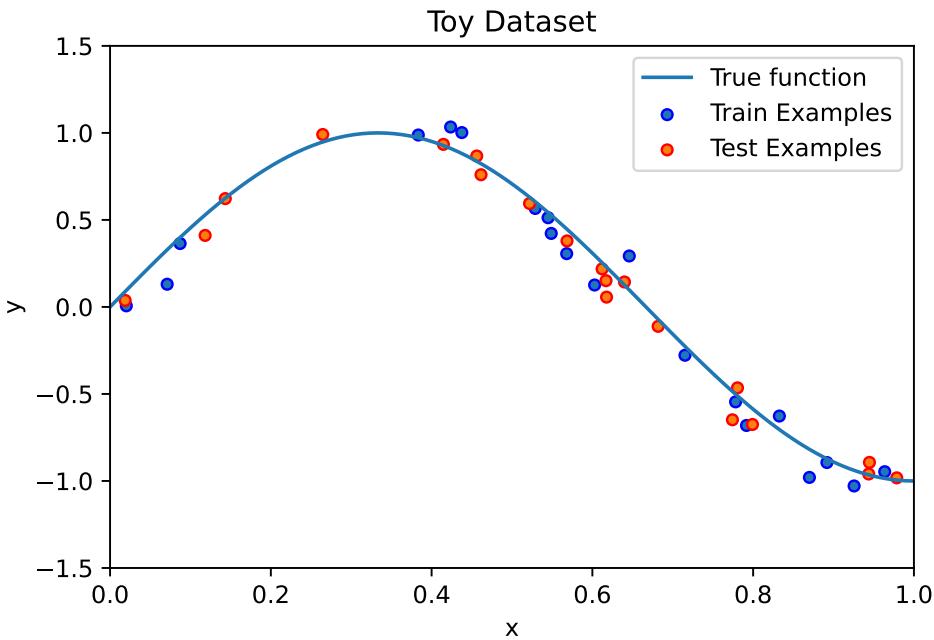


Figure 1.1: Toy dataset for regression.

Code Snippet 1.3: Linear regression.

```

from sklearn.linear_model import LinearRegression

linear_regression = LinearRegression()
linear_regression.fit(X_train[:, np.newaxis], y_train)

plt.plot(x, true_fun(x), label="True function")
plt.scatter(X_train, y_train, edgecolor="b", s=20, label="Train Examples")
plt.plot(x, linear_regression.predict(x[:, np.newaxis]), label="Model")
plt.xlabel("x")
plt.ylabel("y")
plt.xlim((0, 1))
plt.ylim((-1.5, 1.5))

plt.legend(loc="best")
plt.title("Linear regression")
plt.show()

```

You should get a figure similar to Figure 1.2

Task 1: Compute the mean squared error of the learned linear model on the test set.

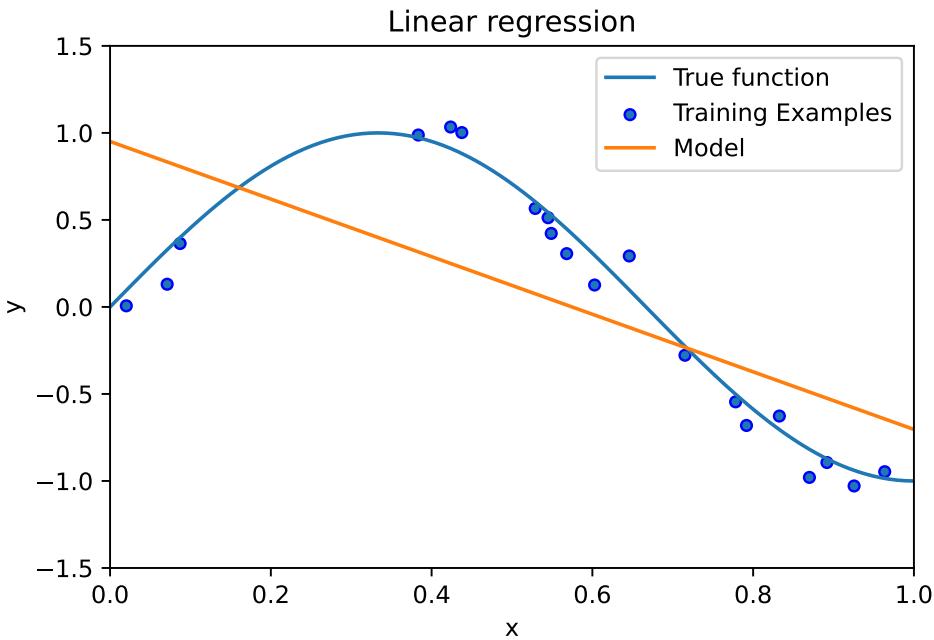


Figure 1.2: Linear regression.

1.1.3 Polynomial regression

The linear model we got in the previous part is too simple to explain the data. In this part we will use more complex models. Let's start with a quadratic function of the form

$$h(x) = w_0 + w_1x + w_2x^2,$$

To get the non-linear basis functions for the quadratic model, we will use *PolynomialFeatures* from *sklearn*

Code Snippet 1.4: Quadratic model.

```
from sklearn.preprocessing import PolynomialFeatures

polynomial_features = PolynomialFeatures(2, include_bias=True)
q_model = LinearRegression()
q_model.fit(polynomial_features.fit_transform(X_train[:, np.newaxis]),
            y_train)
x = np.linspace(0, 1, 100)
plt.plot(x, true_fun(x), label="True function")
plt.scatter(X_train, y_train, edgecolor="b", s=20, label="Samples")
plt.plot(x, q_model.predict(polynomial_features.fit_transform(x[:, np.newaxis])), label="Model")
plt.xlabel("x")
```

```

plt.ylabel("y")
plt.xlim((0, 1))
plt.ylim((-1.5, 1.5))

plt.legend(loc="best")
plt.title("Degree 2")
plt.show()

```

The results will be similar to Figure 1.3

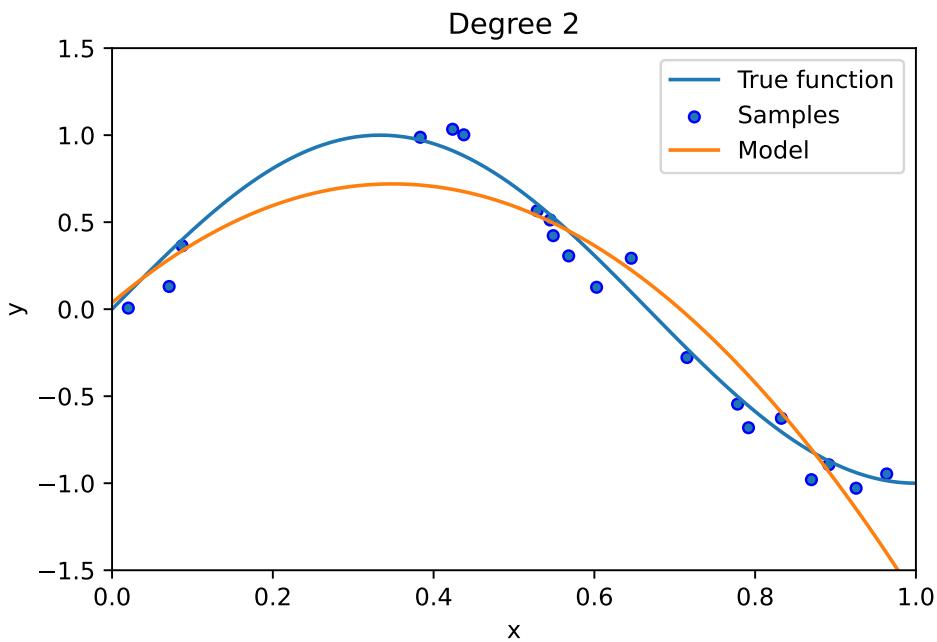


Figure 1.3: Quadratic model for regression.

Task 2: Repeat the previous part with polynomials with degree 4 and 15.

Task 3: Compute the mean squared error of the learned models on the test set. Which model is the best?

1.2 Classification

Classifications is a supervised learning task where the goal is to predict a categorical (discrete) target label. In this part we will experiment with random forests and SVM classifiers on a toy dataset.

1.2.1 Generating examples

In this part, we will generate a toy dataset for classification. We are going to use the `make_classification` function from `sklearn` to generate the data and then split them into training and testing sets using `train_test_split`.

Run the following code to generate and visualize the dataset.

Code Snippet 1.5: Toy dataset for classification.

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

X, y = make_classification(n_samples=500, n_features=2, n_classes=2,
                           n_informative=2, n_redundant=0,
                           n_clusters_per_class=2, random_state=0,
                           shuffle=True, class_sep=1.5)

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.33, random_state=0)

colors = ["b", "r"]
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.set_title("Training Set")
ax2.set_title("Testing Set")
for c in y_train:
    ax1.scatter([X_train[i, 0] for i in np.where(y_train==c)] ,
               [X_train[i, 1] for i in np.where(y_train==c)],
               edgecolor=colors[c], s=20, label="Samples")
    ax2.scatter([X_test[i, 0] for i in np.where(y_test==c)] ,
               [X_test[i, 1] for i in np.where(y_test==c)],
               edgecolor=colors[c], s=20, label="Samples")
```

After running the code above, you should get something similar to Figure 1.4

1.2.2 Classification with random forests

A random forest is an ensemble model that fits a number of decision tree classifiers on various sub-samples of the dataset. Each tree is trained on a bootstrap samples of the training set to introduce randomness in the trees. Furthermore, when selecting a feature for a test node during tree construction, only a subset of the features is considered for the candidate tests. The final prediction in random forests is usually obtained by averaging predictions from all trees.

Let's start by testing a random forest of 2 trees on the dataset we generated in the previous section. The following code trains a random forest on the training set and prints both the training and testing accuracy.

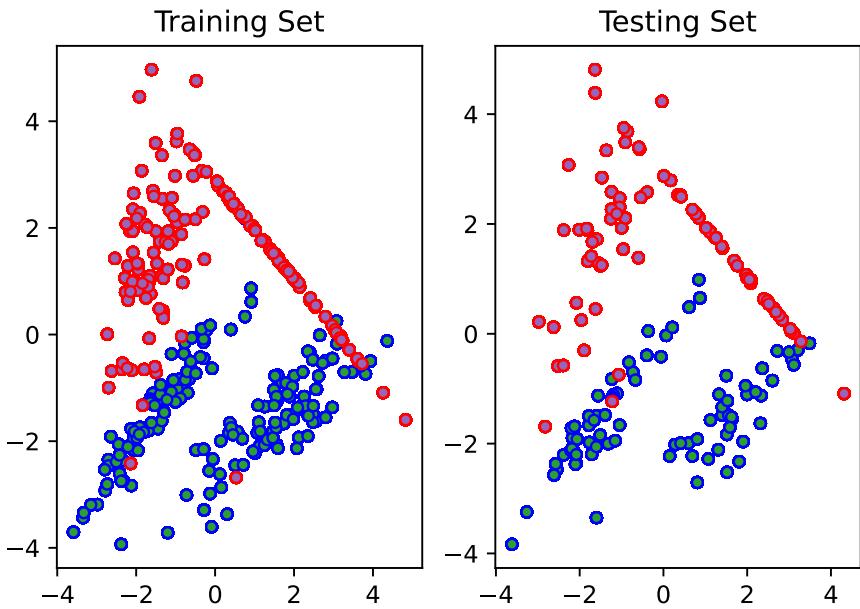


Figure 1.4: Toy dataset for classification.

Code Snippet 1.6: Random forest classifier.

```
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(n_estimators=2,
                             criterion="entropy",
                             max_features="sqrt", max_samples=.8,
                             random_state=0)

clf.fit(X_train, y_train)
print("Training Accuracy: ", clf.score(X_train, y_train))
print("Testing Accuracy: ", clf.score(X_test, y_test))
```

For classification, accuracy is not the only used metrics. There are many other metrics such as precision, recall, F1-score, ...etc. All of these metrics can be derived from the confusion matrix. The following code computes the confusion matrix for the random forest trained in the previous part.

Code Snippet 1.7: Confusion Matrix.

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

y_pred = clf.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                             display_labels=clf.classes_)
disp.plot()
```

Task 4: In the previous example, compute the precision, recall, F1-score. Discuss when accuracy is not the most suitable metric.

Task 5: Train different random forests by changing the number of trees from 2 to 15. Plot the training and testing accuracy of the trained models vs the number of trees. What do you notice?

Decision Surface

Decision surface is a plot that shows how a machine learning model divides the feature space to different class labels. The following code visualizes the decision surface of our random forest model in the previous example. The result should be similar to Figure 1.5

Code Snippet 1.8: Decision Surface.

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.colors import ListedColormap
from sklearn.ensemble import RandomForestClassifier

cmap = plt.cm.RdBu
plot_step = 0.02 # fine step width for decision surface contours

clf = RandomForestClassifier(n_estimators=2, criterion="entropy",
                             max_features="sqrt", max_samples=.8,
                             random_state=0)
clf.fit(X_train, y_train)

# Now plot the decision boundary using a fine mesh as input to a
# filled contour plot
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
xx, yy = np.meshgrid(
    np.arange(x_min, x_max, plot_step),
    np.arange(y_min, y_max, plot_step))

estimator_alpha = 1.0 / len(clf.estimators_)
for tree in clf.estimators_:
    Z = tree.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    cs = plt.contourf(xx, yy, Z, alpha=estimator_alpha, cmap=cmap)

colors = ["r", "b"]
for c in y_train:
    plt.scatter([X_train[i, 0] for i in np.where(y_train==c)] ,
               [X_train[i, 1] for i in np.where(y_train==c)],
               edgecolor=colors[c], s=20, label="Samples")
```

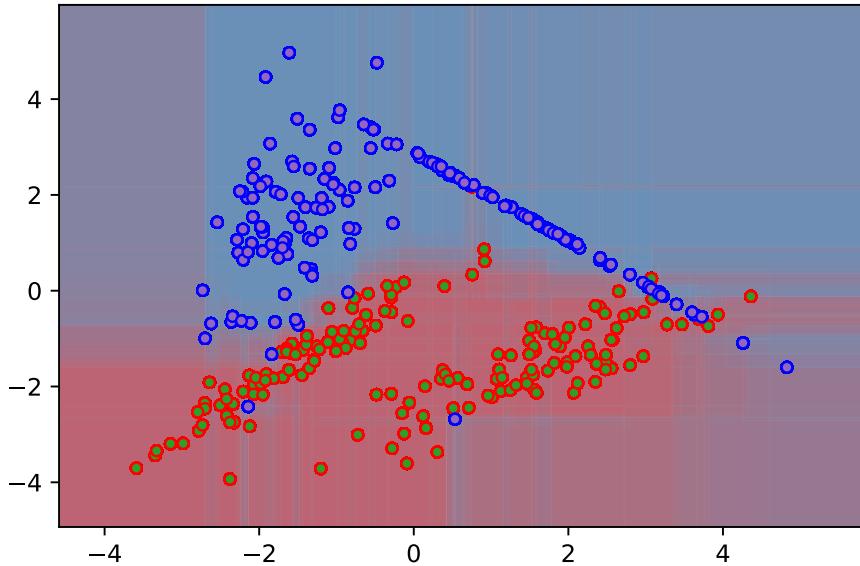


Figure 1.5: Decision Surface.

Task 6: Draw the decision surface for a set of random forests with different number of trees. Do you notice any pattern?

1.2.3 Classification with SVM

In this section, we will test another popular classifier, which is support vector machine (SVM). SVM tries to find a separating hyper-plane between the classes with the maximum margin. The following code trains an SVM classifier with linear kernel on our toy dataset

Code Snippet 1.9: SVM classifier.

```
from sklearn import svm

clf = svm.SVC(kernel="linear", C=100)
clf.fit(X_train, y_train)
print("Training Accuracy: ", clf.score(X_train, y_train))
print("Testing Accuracy: ", clf.score(X_test, y_test))
```

Task 7: Draw the decision surface for the SVM model in the previous example.

Task 8: Train an SVM model with rbf kernel and draw its decision boundary. What do you notice?

Task 9: Repeat task 8 but with C equal 0.5, 1, 100, and 1000. What do you notice?

1.3 Hyper-parameters selection

Hyper-parameters are parameters that are set before starting the training process and not directly learnt within estimators. Typical examples include number of trees and max_depth in random forests, C and kernel for SVM ...etc.

To select hyper-parameters, usually we search the hyper-parameter space for the best cross-validation score (or if the dataset is large enough, we could search for the parameters with the best score on a validation set).

The following code use grid search with cross-validation to find the best hyper-parameters for the SVM example in the previous section.

Code Snippet 1.10: Hyper-parameters selection.

```
from sklearn.model_selection import GridSearchCV
from sklearn import svm

parameters = {"kernel":("linear", "rbf"), "C":[1, 10, 100]}
svc = svm.SVC()

clf = GridSearchCV(svc, parameters, cv=5)
clf.fit(X_train, y_train)
print(clf.best_params_)
```

Task 10: *GridSearchCV* has an attribute called *cv_results_*, which is a dict with keys as column headers and values as columns. Print it and try to explain the values that you get.

1.4 To DO

This part will be given by the instructor during the lab.