ECE 469/ECE 568 Machine Learning

Textbook:
Machine Learning: a Probabilistic Perspective by Kevin Patrick Murphy

Southern Illinois University

September 11, 2024

Training and running a linear model (Scikit-Learn)

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
# Downloading and extracting data
data_link = "https://github.com/ageron/data/raw/main/"
life_satisfac = pd.read_csv(data_link + "lifesat/lifesat.csv")
X = life_satisfac[["GDP per capita (USD)"]].values
Y = life_satisfac[["Life satisfaction"]].values
# Data Visualization
life_satisfac.plot(kind='scatter', grid=True,
x="GDP per capita (USD)", y="Life satisfaction")
plt.axis([25000, 62000, 5, 8])
plt.show()
model = LinearRegression() # Choosing linear model
model.fit(X, Y) # Training the model
# Making the prediction for Cyprus
Cyprus_GDP = [[37655.2]] # Cyprus' GDP per capita in 2020
print(model.predict(Cyprus_GDP)) # Prediction: 6.30165767
```

K-Nearest Neighbors

Previous code can be easily modified to accommodate the K-Nearest Neighbors by replacing the following two lines

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

with the following two lines

```
from sklearn.neighbors import KNeighborsRegressor
model = KNeighborsRegressor(n_neighbors=4)
```

Data Fetching

An example code on fetching and loading data is given below.

```
from pathlib import Path
import pandas as pd
import tarfile
import urllib.request
def load_housing_data():
  tarball_path = Path("datasets/housing.tgz")
  if not tarball_path.is_file():
    Path("datasets").mkdir(parents=True, exist_ok=True)
    url = "https://github.com/ageron/data/raw/main/housing.tgz"
    urllib.request.urlretrieve(url, tarball_path)
    with tarfile.open(tarball_path) as housing_tarball:
      housing_tarball.extractall(path="datasets")
  return pd.read_csv(Path("datasets/housing/housing.csv"))
housing = load_housing_data()
```

Data Fetching

The $info()$ function serves as a valuable tool for obtaining a concise summary of the dataset, including the total row count, the data types of each attribute, and the count of non-null values for each attribute.

```
housing.info()
```

The available categories and the number of items within each category by utilizing the $value\_counts()$ method:

```
housing["ocean_proximity"].value_counts()
```

A summary of the numerical attributed can be obtained via $describe()$ method.

```
housing.describe()
```

Histogram

Plot a histogram for each numerical attribute:

```
import matplotlib.pyplot as plt
housing.hist(bins=40, figsize=(15, 10))
plt.show()
```

Plot a histogram for a specific numerical attribute:

```
import matplotlib.pyplot as plt
housing["longitude"].hist(bins=40, figsize=(6, 4))
plt.show()
```

Test Set Creation

Typically we randomly choose 20% instances of the dataset as the test set.

```
import numpy as np
def shuffle_and_split_data(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

This function can be used as follows.

```
train_set, test_set = shuffle_and_split_data(housing, 0.2)
len(train_set)
len(test_set)
```

This function will create different test data sets each time you run.
Solution 1: Save the test dataset of the first run and load it for the subsequent runs.
Solution 2: Generate the same shuffled indices always by setting the random number generator's seed (eg: with $np.random.seed(42)$) before calling $np.random.permutation()$.

Test Set Creation

Scikit-Learn

```
housing["income_cat"] = pd.cut(housing["median_income"],
bins=[0., 1.5, 3.0, 4.5, 6., np.inf], labels=[1, 2, 3, 4, 5])
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(housing, test_size=0.2,
                                       random_state=42)
```

To generate 10 different stratified splits (for cross validation):

```
from sklearn.model_selection import StratifiedShuffleSplit
splitter = StratifiedShuffleSplit(n_splits=10, test_size=0.2,
                                  random_state=42)
strat_splits = []
for train_index, test_index in splitter.split(housing,
                                              housing["income_cat"]):
    strat_train_set_n = housing.iloc[train_index]
    strat_test_set_n = housing.iloc[test_index]
    strat_splits.append([strat_train_set_n, strat_test_set_n])
```

Use the first split:

```
strat_train_set, strat_test_set = strat_splits[0]
```

Preparing and Cleaning Data

Preparing data:

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Cleaning data:

```
housing.dropna(subset=["total_bedrooms"], inplace=True) # option 1
housing.drop("total_bedrooms", axis=1) # option 2
median = housing["total_bedrooms"].median() # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

Instead of the above 3 options we can use the Scikit-Learn class: SimpleImputer.

Feature Scaling

Selecting the numerical data types:

```
housing_num = housing.select_dtypes(include=[np.number])
```

Normalization:

```
from sklearn.preprocessing import MinMaxScaler
min_max_scaler = MinMaxScaler(feature_range=(-1, 1))
housing_num_normalized = min_max_scaler.fit_transform(housing_num)
```

Standardization: Subtracts the mean and divides by the standard deviation.

```
from sklearn.preprocessing import StandardScaler
std_scaler = StandardScaler()
housing_num_standadized = std_scaler.fit_transform(housing_num)
```

Evaluation

Linear Regression:

```
from sklearn.linear_model import LinearRegression
lin_reg = make_pipeline(preprocessing, LinearRegression())
lin_reg.fit(housing, housing_labels)
```

Check the output:

```
housing_predictions = lin_reg.predict(housing)
housing_predictions[:5].round(-2) # -2=rounded to the nearest 100
housing_labels.iloc[:5].values
```

Decision Tree:

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = make_pipeline(preprocessing,
                         DecisionTreeRegressor(random_state=42))
tree_reg.fit(housing, housing_labels)
```

Check the output:

```
housing_predictions = tree_reg.predict(housing)
tree_rmse = mean_squared_error(housing_labels,housing_predictions,
                               squared=False)
```

Cross Validation Evaluation

```
from sklearn.model_selection import cross_val_score
tree_rmses = -cross_val_score(tree_reg, housing, housing_labels,
                              scoring="neg_root_mean_squared_error", cv=10)
```

Check the output:

```
d.Series(tree_rmses).describe()
```

RandomForestRegressor:

```
from sklearn.ensemble import RandomForestRegressor
forest_reg = make_pipeline(preprocessing,
                           RandomForestRegressor(random_state=42))
forest_rmses= -cross_val_score(forest_reg, housing,housing_labels,
                               scoring="neg_root_mean_squared_error", cv=10)
```

Check the output:

```
pd.Series(forest_rmses).describe()
```

Linear regression

Generate noisy data:

```
import numpy as np
np.random.seed(42) # to make the code reproducible
n = 200 # number of instances
X = 2 * np.random.rand(n, 1) # input vector
Y = 4 + 3 * X + np.random.randn(n, 1) # output vector
```

Calculate weights:

```
from sklearn.preprocessing import add_dummy_feature
X0 = add_dummy_feature(X) # add x0 = 1 to each instance
weights = np.linalg.inv(X0.T @ X0) @ X0.T @ Y
```

Pseudo-inverse

```
np.linalg.pinv(X0) @ y
```

Linear regression

Making predictions:

```
new_X = np.array([[0], [2]])
new_X0 = add_dummy_feature(new_X) # add x0 = 1 to each instance
new_Y = new_X0 @ weights
new_Y
```

Plotting the data:

```
import matplotlib.pyplot as plt
plt.plot(new_X, new_Y, "r-", label="Predictions")
plt.plot(X, Y, "b.")
plt.show()
```

Linear regression

Scikit-Learn:

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, Y)
lin_reg.coef_, lin_reg.intercept_
lin_reg.predict(new_X)
```

The feature weights ($coeff\_$) and the bias term ($intercept\_$) are separately given by Scikit-Learn.

Gradient Descent

```
alpha = 0.1 # learning rate
n_iter = 1000
n_inst = len(X0) # number of instances

np.random.seed(42)
weights = np.random.randn(2, 1) # random initialization of weights

for i in range(n_iter):
    grad = 2 / n_inst * X0.T @ (X0 @ weights - Y)
    weights = weights - alpha * grad
```

Check the weights:

```
weights
```

Stochastic Gradient Descent

```
n_iter = 1000
n_inst = len(X0) # number of instances

s0, s1 = 5, 50 # learning schedule hyperparameters

def learning_schedule(s):
    return s0 / (s + s1)

np.random.seed(42)
weights = np.random.randn(2, 1) # random initialization of weights

for i in range(n_iter):
    for j in range(n_inst):
        rand_idx = np.random.randint(n_inst)
        xj = X0[rand_idx : rand_idx + 1]
        yj = Y[rand_idx : rand_idx + 1]
        grad = 2 * xj.T @ (xj @ weights - yj)
        alpha = learning_schedule(i * n_inst + j)
        weights = weights - alpha * grad
```

Stochastic Gradient Descent

Scikit-Learn:

```
from sklearn.linear_model import SGDRegressor

sgd_rg=SGDRegressor(max_iter=1000,tol=1e-5,penalty=None,eta0=0.01,
                    n_iter_no_change=100, random_state=42)

sgd_rg.fit(X,Y.ravel()) #Y.ravel() since fit() expects 1D targets
```

Check the solution:

```
sgd_rg.intercept_, sgd_rg.coef_
```

Polynomial Regression

```
np.random.seed(42)
n = 200
X = 6 * np.random.rand(n, 1) - 3
Y = 0.5 * X ** 2 + X + 2 + np.random.randn(n, 1)
```

Use Scikit-Learn
Add the square of each feature as a new feature.

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_p = poly_features.fit_transform(X)
X[0], X_p[0]
```

X_p contains X and the square of X.
Use linear regression assuming there are 2 features in this example.

```
lin_reg = LinearRegression()
lin_reg.fit(X_p, Y)
lin_reg.intercept_, lin_reg.coef_
```

Logistic Regression

Develop a classifier to identify the Iris virginica type based only on the petal width.

Load data:

```
from sklearn.datasets import load_iris
iris = load_iris(as_frame=True)
```

Create test and training datasets and train the logistic regression model.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
X = iris.data[["petal width (cm)"]].values
Y = iris.target_names[iris.target] == 'virginica'
Xtrain,Xtest,Ytrain,Ytest=train_test_split(X,Y,random_state=42)
log_reg = LogisticRegression(random_state=42)
log_reg.fit(Xtrain, Ytrain)
```

Logistic Regression

Petal width versus probability graph:

```
X_axis = np.linspace(0, 3, 1000).reshape(-1, 1)
# reshape to get a column vector for petal widths

Y_axis = log_reg.predict_proba(X_axis)

DB = X_axis[Y_axis[:, 1] >= 0.5][0, 0] # desicion boundary

plt.plot(X_axis, Y_axis[:, 0], "b--", linewidth=2,
         label="Not Iris virginica proba")
plt.plot(X_axis, Y_axis[:, 1], "g-", linewidth=2,
         label="Iris virginica proba")
plt.plot([DB, DB], [0, 1], "k:", linewidth=2,
         label="Decision boundary")
plt.show()

DB
log_reg.predict([[1.7], [1.5]])
```

Softmax Regression

Let's use softmax regression to classify the iris plants into all three classes.

```
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
Y = iris["target"]
Xtrain,Xtest,Ytrain,Ytest=train_test_split(X,Y,random_state=42)

softmax_reg = LogisticRegression(C=30, random_state=42)
softmax_reg.fit(Xtrain, Ytrain)
```

New input: petal which is 5 cm long and 2 cm wide

```
softmax_reg.predict([[5, 2]])
softmax_reg.predict_proba([[5, 2]]).round(2)
```

Answer: class 2 - Iris virginica - 96% probability

More Examples

https://scikit-learn.org/stable/auto_examples/index.html