Machine Learning in practice with sklearn

Key ML Terminology

Labels

A label is the thing we're predicting—the `y` variable in simple linear regression. The label could be the future price of wheat, the kind of animal shown in a picture, the meaning of an audio clip, or just about anything.

Features

A feature is an input variable—the `x` variable in simple linear regression. A simple machine learning project might use a single feature, while a more sophisticated machine learning project could use millions of features, specified as: $x_1, x_2, \ldots, x_n$

In the spam detector example, the features could include the following:

- words in the email text
- sender's address
- time of day the email was sent
- email contains the phrase "one weird trick."

Example

For example, the following table shows 5 labeled examples from a data set containing information about housing prices in California:

housingMedianAge (feature)	totalRooms (feature)	totalBedrooms (feature)	medianHouseValue (label)
15	12	6	66900
19	9	4	80100
17	8	3	85700
14	5	2	73400

Once we've trained our model with labeled examples, we use that model to predict the label on unlabeled examples.

Regression vs. classification

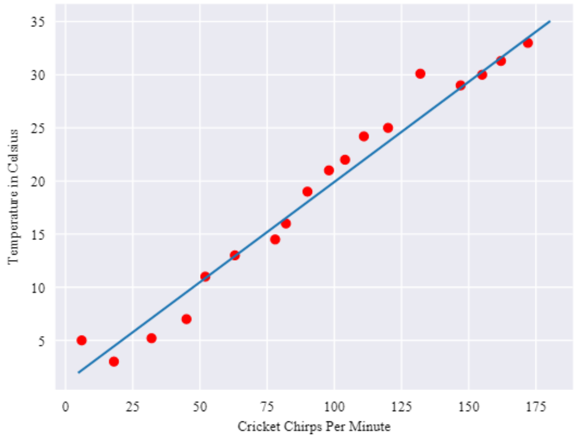
A **regression** model predicts continuous values. For example, regression models make predictions that answer questions like the following:

- What is the value of a house in California?
- What is the probability that a user will click on this ad?

A **classification** model predicts discrete values. For example, classification models make predictions that answer questions like the following:

- Is a given email message spam or not spam?
- Is this an image of a dog, a cat, or a hamster?

Descending into ML: Linear Regression



A linear relationship.

True, the line doesn't pass through every dot, but the line does clearly show the relationship between chirps and temperature. Using the equation for a line, you could write down this relationship as follows:

$$y = mx + b$$

- y is the temperature in Celsius—the value we're trying to predict.
- *m* is the slope of the line.
- x is the number of chirps per minute—the value of our input feature.
- b is the y-intercept (an (x,y) point with x=0).

By convention in machine learning, you'll write the equation for a model slightly differently:

$$y' = b + w_1 x_1$$

- y' is the predicted label (a desired output).
- b is the bias (the y-intercept), sometimes referred to as w_0.
- w_1 is the weight of feature 1. Weight is the same concept as the "slope" in the traditional equation of a line.
- x_1 is a feature (a known input).

Although this model uses only one feature, a more sophisticated model might rely on multiple features $y' = b + w_1 x_1 + w_2 x_2 + w_3 x_3$

1.

Linear Regression

Exercise 1

Open Colab

Start with Exercise 1.
Choose a dataset for regression, load the dataset, explore the values

Regression – boston housing dataset

```
 import pandas as pd
 import numpy as np
 from sklearn.datasets import load_boston
 boston = load_boston()
 print(boston)
# "Transform data set to data frame"
# data = the data we want on the independent variables also known as the x values
# feature_names  = column names of data
# target = the target variable

df_x = pd.DataFrame(boston.data, columns=boston.feature_names)
df_y = pd.DataFrame(boston.target )
```

Regression – boston housing dataset

```
#some statistics

df_x.describe()
#initialize the linear regression model

reg = linear_model.LinearRegression()
```

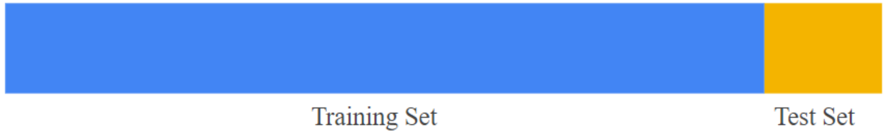
Training and Test Sets: Splitting Data

A **test** set is a data set used to evaluate the model **developed from a training set**.

Divide your data set into two subsets:

- **training** set—a subset to train a model.
- **test** set—a subset to test the trained model.

You could imagine slicing the single data set as follows:



Never train on test data. If you are seeing surprisingly good results on your evaluation metrics, it might be a sign that you are accidentally training on the test set. For example, high accuracy might indicate that test data has leaked into the training set.

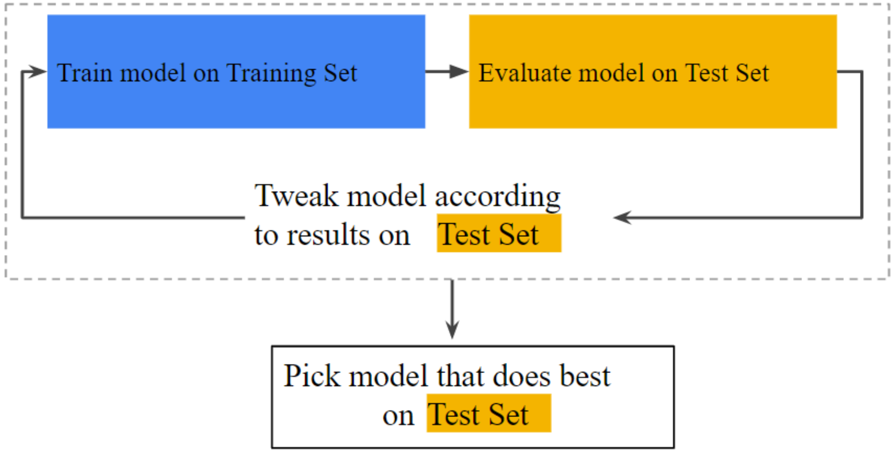
Regression – boston housing dataset

```
#split the data 67% training and 33% testing data


x_train, x_test , y_train, y_test = train_test_split(df_x, df_y,
test_size=0.33, random_state=42)
```

Validation Set: Another Partition

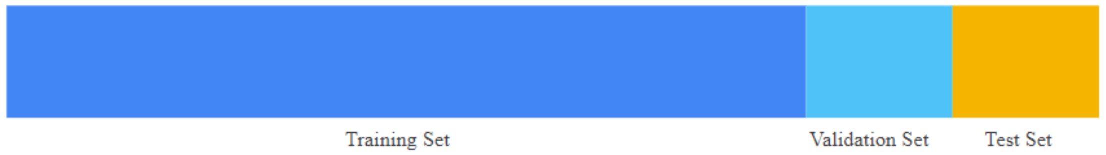
The previous module introduced partitioning a data set into a training set and a test set. This partitioning enabled you to train on one set of examples and then to test the model against a different set of examples. With two partitions, the workflow could look as follows:


Train model on Training Set
Evaluate model on Test Set
Tweak model according to results on Test Set
Pick model that does best on Test Set

A possible workflow?

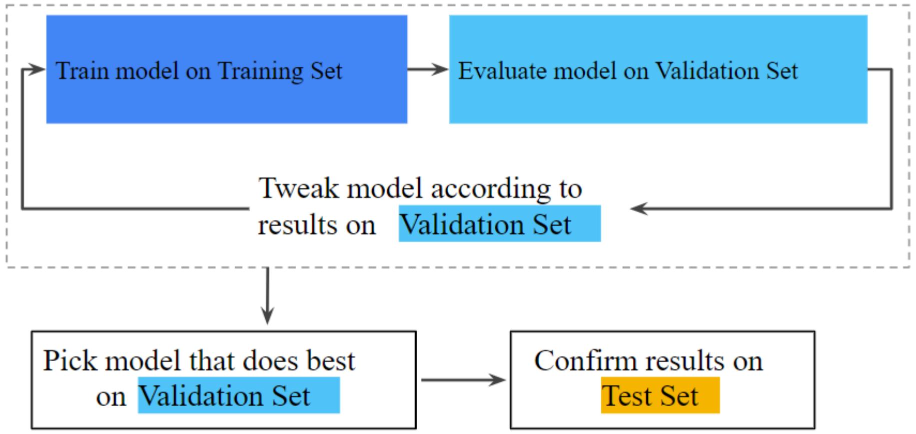
The previous module introduced partitioning a data set into a training set and a test set. This partitioning enabled you to train on one set of examples and then to test the model against a different set of examples. With two partitions, the, "Tweak model" means adjusting anything about the model you can dream up—from changing the learning rate, to adding or removing features, to designing a completely new model from scratch. At the end of this workflow, you pick the model that does best on the test set.

Dividing the data set into two sets is a good idea, but not a panacea. You can greatly reduce your chances of overfitting by partitioning the data set into the three subsets shown in the following figure: workflow could look as follows:



Validation Set: Another Partition

Use the validation set to evaluate results from the training set. Then, use the test set to double-check your evaluation after the model has "passed" the validation set. The following figure shows this new workflow:


Train model on Training Set
Evaluate model on Validation Set
Tweak model according to results on Validation Set
Pick model that does best on Validation Set
Confirm results on Test Set

How to obtain a validation set using `train_test_split`

```
# set aside 20% of train and test data for evaluation
X_train, X_test, y_train, y_test = train_test_split(train, test,
    test_size=0.2, shuffle = True, random_state = 8)

# Use the same function above for the validation set
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
    test_size=0.15, random_state= 8)


print("X_train shape: {}".format(X_train.shape))
print("X_test shape: {}".format(X_test.shape))
print("y_train shape: {}".format(y_train.shape))
print("y_test shape: {}".format(y_test.shape))
print("X_val shape: {}".format(y_train.shape))
print("y val shape: {}".format(y_test.shape))
```

Next step: train the model

```
#train the model with our training data


reg.fit(x_train, y_train)
```

Training and Loss

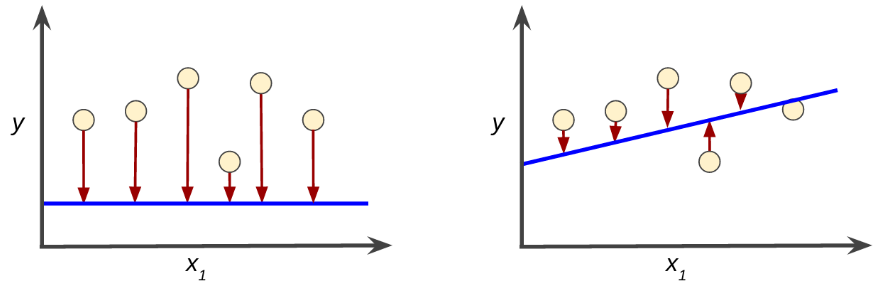
Training a model simply means learning (determining) good values for all the weights and the bias from labeled examples. In supervised learning, a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss; this process is called ***empirical risk minimization***.

Loss is the penalty for a bad prediction. That is, loss is a number indicating how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater. The goal of training a model is to find a set of weights and biases that have low loss, on average, across all examples.

The arrows represent loss.

The blue lines represent predictions

High loss in the left model; low loss in the right model.



Next step: train the model

```
# print the coefficience/weights for each feature/column of our model
print(reg.coef_)
# print the predictions on test data
y_pred = reg.predict(x_test)
print(y_pred)
# print the actual values

print(y_test)


# check the performance MSE
print( np.mean( (y_pred - y_test)**2 ))
# check the performance MSE and sklearn.metrics

from sklearn.metrics import mean_squared_error
print ( mean_squared_error (y_test, y_pred))
```

2.

Classification

Exercise 1

Classification

Choose a dataset for classification, load the dataset, perform classification using any classifier from sklearn library

An Aesop's Fable: The Boy Who Cried Wolf

A shepherd boy gets bored tending the town's flock. To have some fun, he cries out, "Wolf!" even though no wolf is in sight. The villagers run to protect the flock, but then get really mad when they realize the boy was playing a joke on them.

[Iterate previous paragraph N times.]

One night, the shepherd boy sees a real wolf approaching the flock and calls out, "Wolf!" The villagers refuse to be fooled again and stay in their houses. The hungry wolf turns the flock into lamb chops. The town goes hungry. Panic ensues.

Let's make the following definitions:

"**Wolf**" is a positive class.
"**No wolf**" is a negative class.

"**Wolf**" is a positive class.
"**No wolf**" is a negative class.

Confusion matrix

True Positive

Reality: A wolf threatened.

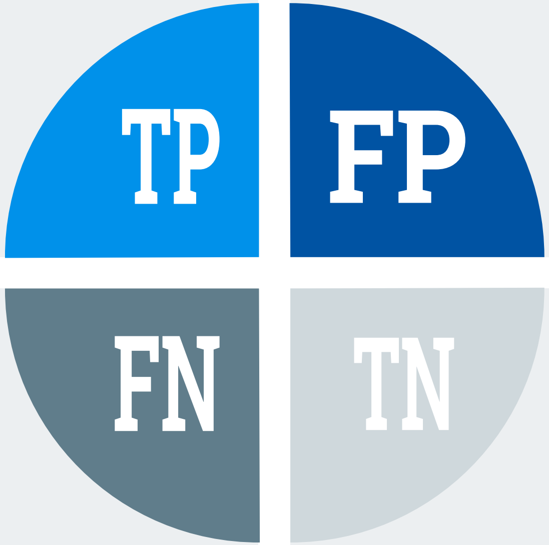
Shepherd said: "Wolf."

Outcome: Shepherd is a hero.

False Positive

Reality: No wolf threatened.

Shepherd said: "Wolf.“
Outcome: Villagers are angry at shepherd for waking them up.



Reality: A wolf threatened.

Shepherd said: "No wolf."

Outcome: The wolf ate all the sheep.

False Negative

Reality: No wolf threatened.
Shepherd said: "No wolf."
Outcome: Everyone is fine.
True Negative

*A **true positive** is an outcome where the model correctly predicts the positive class.*

*Similarly, a **true negative** is an outcome where the model correctly predicts the negative class.*

*A **false positive** is an outcome where the model incorrectly predicts the positive class.*

*And a **false negative** is an outcome where the model incorrectly predicts the negative class.*

Accuracy

Informally, accuracy is the fraction of predictions our model got right. Formally, accuracy has the following definition:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

For binary classification, accuracy can also be calculated in terms of positives and negatives as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Accuracy

Let's try calculating accuracy for the following model that classified 100 tumors as malignant (the positive class) or benign (the negative class):

True Positive (TP): Reality: Malignant ML model predicted: Malignant Number of TP results: 1	False Positive (FP): Reality: Benign ML model predicted: Malignant Number of FP results: 1
False Negative (FN): Reality: Malignant ML model predicted: Benign Number of FN results: 8	True Negative (TN): Reality: Benign ML model predicted: Benign Number of TN results: 90

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{1 + 90}{1 + 90 + 1 + 8} = 0.91$$

Accuracy comes out to 0.91, or 91% (91 correct predictions out of 100 total examples). That means our tumor classifier is doing a great job of identifying malignancies, right?

Accuracy

Actually, let's do a closer analysis of positives and negatives to gain more insight into our model's performance.

Of the 100 tumor examples, 91 are benign (90 TNs and 1 FP) and 9 are malignant (1 TP and 8 FNs).

Of the 91 benign tumors, the model correctly identifies 90 as benign. That's good. However, of the 9 malignant tumors, the model only correctly identifies 1 as malignant—a terrible outcome, as 8 out of 9 malignancies go undiagnosed!

While 91% accuracy may seem good at first glance, another tumor-classifier model that always predicts benign would achieve the exact same accuracy (91/100 correct predictions) on our examples. In other words, our model is no better than one that has zero predictive ability to distinguish malignant tumors from benign tumors.

Accuracy alone doesn't tell the full story when you're working with a class-imbalanced data set, like this one, where there is a significant disparity between the number of positive and negative labels.

Precision

Precision attempts to answer the following question:

What proportion of positive identifications was actually correct?

Precision is defined as follows:

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

Recall

Recall attempts to answer the following question:

What proportion of actual positives was identified correctly?

Mathematically, recall is defined as follows:

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

Instructions for use

Let's calculate precision for our tumor classifier:

True Positives (TPs): 1	False Positives (FPs): 1
False Negatives (FNs): 8	True Negatives (TNs): 90

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{1}{1+1} = 0.5$$

Our model has a precision of 0.5—in other words, when it predicts a tumor is malignant, it is correct 50% of the time.

Let's calculate recall for our tumor classifier:

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{1}{1+8} = 0.11$$

Our model has a recall of 0.11—in other words, it correctly identifies 11% of all malignant tumors.

“

To fully evaluate the effectiveness of a model, you must examine both precision and recall. ***Unfortunately, precision and recall are often in tension.*** *That is, improving precision typically reduces recall and vice versa.*

Various metrics have been developed that rely on both precision and recall. For example, see F1-score https://en.wikipedia.org/wiki/F-score

Next step: prepare the model

```
 import numpy as np
 import pandas as pd
 import matplotlib.pyplot as plt
 from sklearn.datasets import load_breast_cancer
 from sklearn.model_selection import train_test_split
 plt.style.use('ggplot')

 # Breast cancer dataset for classification
 breast_cancer_data = load_breast_cancer()
 print (breast_cancer_data .feature_names)
 print (breast_cancer_data .target_names)
 # put the data in a dataframe
df_x = pd.DataFrame(breast_cancer_data.data, columns= breast_cancer_data.feature_names)
df_y = pd.DataFrame(breast_cancer_data.target )
df_x.info()
X_train, X_test, y_train, y_test = train_test_split(df_x, df_y, test_size = 0.2, random_state
= 42)
```

Next step: load the classifier and train the model

```
#train the model with our training data

from sklearn.naive_bayes import GaussianNB


nbclf = GaussianNB().fit(X_train, y_train)

predicted = nbclf.predict(X_test)

print('Breast cancer dataset')

print('Accuracy of GaussianNB classifier on training set: {:.2f}'.format(nbclf.score(X_train,
y_train)))

print('Accuracy of GaussianNB classifier on test set: {:.2f}'.format(nbclf.score(X_test, y_test)))
```

Next step: load the classifier and train the model

```
from sklearn import metrics


print("Classification report for classifier %s:\n%s\n"
      % (nbclf, metrics.classification_report(y_test, predicted)))
print("Confusion matrix:\n%s" % metrics.confusion_matrix(y_test, predicted))
```

Next step: load the classifier and train the model

```
from sklearn.neighbors import KNeighborsClassifier

clf = KNeighborsClassifier()
clf.fit(X_train, y_train)
prediction = clf.predict(X_test)

print('Breast cancer dataset')
print('Accuracy of GaussianNB classifier on training set: {:.2f}'.format(clf.score(X_train, y_train)))
print('Accuracy of GaussianNB classifier on test set: {:.2f}'.format(clf.score(X_test, y_test)))

print("\n Classification report for classifier %s:\n%s\n"
      % (clf, metrics.classification_report(y_test, prediction)))
print("Confusion matrix:\n%s" % metrics.confusion_matrix(y_test, prediction))
```