

Machine learning with scikit-learn

Gyarmathy Timea

May 21, 2018

Contents

1	Overview	1
1.1	Running instructions	2
1.2	Theoretical aspects	2
1.2.1	Machine learning	2
1.2.2	Training set and testing set	3
1.3	Existing Example	3
1.3.1	Loading an example dataset	3
1.3.2	Learning and predicting	4
1.4	Following a tutorial step by step	6
1.4.1	Load the data	6
1.4.2	Summarize the dataset	7
1.4.3	Data visualization	8
1.4.4	Evaluate Some Algorithms	9
1.4.5	Make Predictions	12
2	Proposed problem	14
2.1	Specification	14
2.1.1	The dataset	14
2.1.2	Project steps	14
2.2	Implementation	15
2.2.1	The images	15
2.2.2	Testing algorithms	15
2.2.3	The Algorithm	18
2.2.4	Results	23
2.3	Conclusions	24
2.4	Further developments	24

1 Overview

Scikit-learn (formerly scikits.learn) is a free software machine learning library for the Python programming language.[1] It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

The scikit-learn project started as scikits.learn, a Google Summer of Code project by David Cournapeau. Its name stems from the notion that it is a



Figure 1: The scikit-learn logo

”SciKit” (SciPy Toolkit), a separately-developed and distributed third-party extension to SciPy. The original codebase was later rewritten by other developers. In 2010 Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort and Vincent Michel, all from INRIA took leadership of the project and made the first public release on February the 1st 2010. Of the various scikits, scikit-learn as well as scikit-image were described as ”well-maintained and popular” in November 2012.

As of 2017, scikit-learn is under active development.[2]

1.1 Running instructions

Scikit-learn requires:

- Python ($i=2.7$ or $i=3.3$),
- NumPy ($i=1.8.2$),
- SciPy ($i=0.13.3$).

If you already have a working installation of numpy and scipy, the easiest way to install scikit-learn is using pip

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

or conda:

```
conda install scikit-learn
```

Canopy and Anaconda both ship a recent version of scikit-learn, in addition to a large set of scientific python library for Windows, Mac OSX and Linux. Anaconda offers scikit-learn as part of its free distribution. To download Anaconda, you have to access: <https://www.anaconda.com/download/> After you have downloaded, it, you can work from the **Anaconda Prompt** with scikit-learn.

To work with scikit-learn, you have to open the installed Anaconda Prompt, and type in ”python” to open the python console.

1.2 Theoretical aspects

1.2.1 Machine learning

In general, a learning problem considers a set of n samples of data and then tries to predict properties of unknown data. If each sample is more than a

single number and, for instance, a multi-dimensional entry (aka multivariate data), it is said to have several attributes or features.

We can separate learning problems in a few large categories:

- supervised learning, in which the data comes with additional attributes that we want to predict. This problem can be either:
 - classification: samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data. An example of classification problem would be the handwritten digit recognition example, in which the aim is to assign each input vector to one of a finite number of discrete categories. Another way to think of classification is as a discrete (as opposed to continuous) form of supervised learning where one has a limited number of categories and for each of the n samples provided, one is to try to label them with the correct category or class.
 - regression: if the desired output consists of one or more continuous variables, then the task is called regression. An example of a regression problem would be the prediction of the length of a salmon as a function of its age and weight.
- unsupervised learning, in which the training data consists of a set of input vectors x without any corresponding target values. The goal in such problems may be to discover groups of similar examples within the data, where it is called clustering, or to determine the distribution of data within the input space, known as density estimation, or to project the data from a high-dimensional space down to two or three dimensions for the purpose of visualization.

1.2.2 Training set and testing set

Machine learning is about learning some properties of a data set and applying them to new data. This is why a common practice in machine learning to evaluate an algorithm is to split the data at hand into two sets, one that we call the training set on which we learn data properties and one that we call the testing set on which we test these properties.

1.3 Existing Example

1.3.1 Loading an example dataset

scikit-learn comes with a few standard datasets, for instance the iris and digits datasets for classification and the boston house prices dataset for regression.

In the following, we start a Python interpreter from our shell and then load the iris and digits datasets. Our notational convention is that $\$$ denotes the shell prompt while $>>>$ denotes the Python interpreter prompt:

```
 $\$$  python
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> digits = datasets.load_digits()
```

A dataset is a dictionary-like object that holds all the data and some meta-data about the data. This data is stored in the `.data` member, which is a `n_samples, n_features` array. In the case of supervised problem, one or more response variables are stored in the `.target` member. More details on the different datasets can be found in the dedicated section.

For instance, in the case of the digits dataset, `digits.data` gives access to the features that can be used to classify the digits samples:

```
>>> print(digits.data)
[[ 0.  0.  5. ..., 0.  0.  0.]
 [ 0.  0.  0. ..., 10.  0.  0.]
 [ 0.  0.  0. ..., 16.  9.  0.]
 ...,
 [ 0.  0.  1. ..., 6.  0.  0.]
 [ 0.  0.  2. ..., 12.  0.  0.]
 [ 0.  0. 10. ..., 12.  1.  0.]]
```

and `digits.target` gives the ground truth for the digit dataset, that is the number corresponding to each digit image that we are trying to learn:

```
>>> digits.target
array([0, 1, 2, ..., 8, 9, 8])
```

Shape of the data arrays

The data is always a 2D array, shape `(n_samples, n_features)`, although the original data may have had a different shape. In the case of the digits, each original sample is an image of shape `(8, 8)` and can be accessed using:

```
>>> digits.images[0]
array([[ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.],
       [ 0.,  0., 13., 15., 10., 15.,  5.,  0.],
       [ 0.,  3., 15.,  2.,  0., 11.,  8.,  0.],
       [ 0.,  4., 12.,  0.,  0.,  8.,  8.,  0.],
       [ 0.,  5.,  8.,  0.,  0.,  9.,  8.,  0.],
       [ 0.,  4., 11.,  0.,  1., 12.,  7.,  0.],
       [ 0.,  2., 14.,  5., 10., 12.,  0.,  0.],
       [ 0.,  0.,  6., 13., 10.,  0.,  0.,  0.]])
```

The simple example on this dataset illustrates how starting from the original problem one can shape the data for consumption in scikit-learn.

1.3.2 Learning and predicting

In the case of the digits dataset, the task is to predict, given an image, which digit it represents. We are given samples of each of the 10 possible classes (the digits zero through nine) on which we fit an estimator to be able to predict the classes to which unseen samples belong.

In scikit-learn, an estimator for classification is a Python object that implements the methods `fit(X, y)` and `predict(T)`.

An example of an estimator is the class `sklearn.svm.SVC` that implements support vector classification. The constructor of an estimator takes as arguments the parameters of the model, but for the time being, we will consider the estimator as a black box:

```
>>> from sklearn import svm
>>> clf = svm.SVC(gamma=0.001, C=100.)
```

Choosing the parameters of the model:

In this example we set the value of `gamma` manually. It is possible to automatically find good values for the parameters by using tools such as grid search and cross validation.

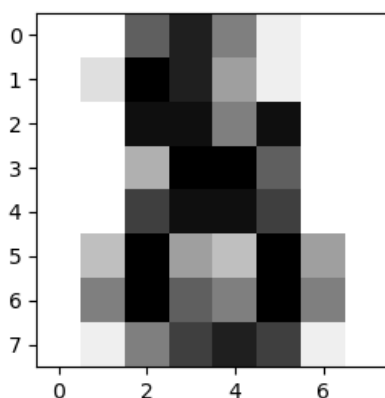
We call our estimator instance `clf`, as it is a classifier. It now must be fitted to the model, that is, it must learn from the model. This is done by passing our training set to the `fit` method. As a training set, let us use all the images of our dataset apart from the last one. We select this training set with the `[:-1]` Python syntax, which produces a new array that contains all but the last entry of `digits.data`:

```
>>> clf.fit(digits.data[:-1], digits.target[:-1])
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

Now you can predict new values, in particular, we can ask to the classifier what is the digit of our last image in the `digits` dataset, which we have not used to train the classifier:

```
>>> clf.predict(digits.data[-1:])
array([8])
```

The corresponding image is the following:



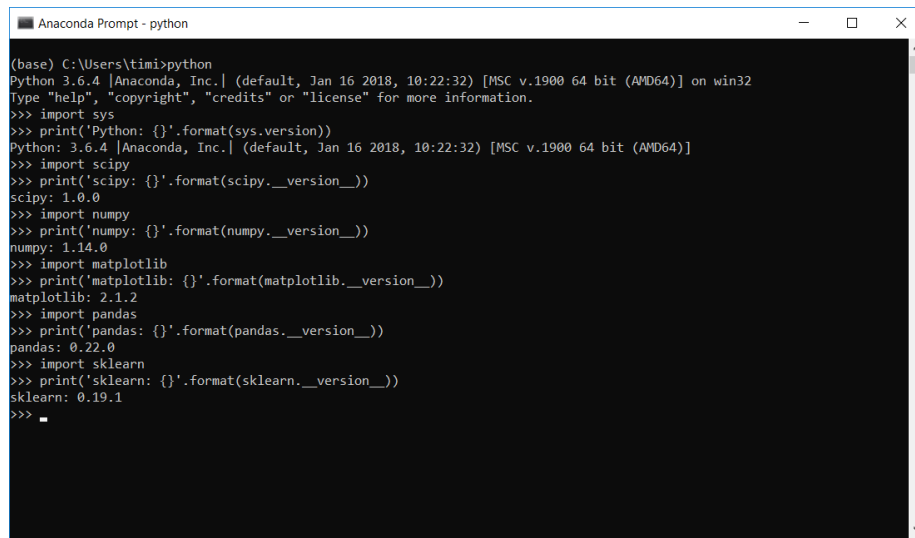
As you can see, it is a challenging task: the images are of poor resolution.

1.4 Following a tutorial step by step

1.4.1 Load the data

The best small project to start with on a new tool is the classification of iris flowers (e.g. the iris dataset we loaded).[3]

First I set up the environment by opening the Anaconda Prompt and typing in python. Then, I import the libraries I'll work with (Figure 2).



```
(base) C:\Users\timi>python
Python 3.6.4 [Anaconda, Inc.] (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> print('Python: {}'.format(sys.version))
Python: 3.6.4 [Anaconda, Inc.] (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)]
>>> import scipy
>>> print('scipy: {}'.format(scipy.__version__))
scipy: 1.0.0
>>> import numpy
>>> print('numpy: {}'.format(numpy.__version__))
numpy: 1.14.0
>>> import matplotlib
>>> print('matplotlib: {}'.format(matplotlib.__version__))
matplotlib: 2.1.2
>>> import pandas
>>> print('pandas: {}'.format(pandas.__version__))
pandas: 0.22.0
>>> import sklearn
>>> print('sklearn: {}'.format(sklearn.__version__))
sklearn: 0.19.1
>>> _
```

Figure 2: Setting up the environment

As you can see, I also printed the versions to make sure everything fine.

I am using the iris flowers dataset and load it from a CSV file URL. This dataset is famous because it is used as the hello world dataset in machine learning and statistics by pretty much everyone.

The dataset contains 150 observations of iris flowers. There are four columns of measurements of the flowers in centimeters. The fifth column is the species of the flower observed. All observed flowers belong to one of three species.

Now, as I follow a tutorial, I import all the libraries I am going to work with by copy-pasting in the following code into the console:

```
import pandas
from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt
from sklearn import model_selection
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
```

```
from sklearn.svm import SVC
```

A machine learning project has a number of well known steps:

1. Define Problem.
2. Prepare Data.
3. Evaluate Algorithms.
4. Improve Results.
5. Present Results.

In our case the problem is to observe the features of the flowers and what is specific for every species and to predict which species the flower belongs to.

In the previous example, we loaded the datasets from the examples shipped with scikit-learn. Here is how we are going to do it now, from a CSV file, as the project will use a dataset saved in this format:

```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
dataset = pandas.read_csv(url, names=names)
```

1.4.2 Summarize the dataset

Now it is time to take a look at the data.

In this step we are going to take a look at the data a few different ways:

1. Dimensions of the dataset.
2. Peek at the data itself.
3. Statistical summary of all attributes.
4. Breakdown of the data by the class variable.

Dimensions of Dataset: We can get a quick idea of how many instances (rows) and how many attributes (columns) the data contains with the shape property.

```
print(dataset.shape)
```

Peek at the Data: It is also always a good idea to actually eyeball your data.

```
print(dataset.head(20))
```

Statistical Summary: This includes the count, mean, the min and max values as well as some percentiles.

```
print(dataset.describe())
```

Class Distribution: Lets now take a look at the number of instances (rows) that belong to each class. We can view this as an absolute count.

```
print(dataset.groupby('class').size())
```

The result I got by running the commands can be seen in Figure 3. At the statistical summary we can see that all of the numerical values have the same scale (centimeters) and similar ranges between 0 and 8 centimeters. At the class distribution we can observe how good the dataset is, each class having the same number of instances (50 or 33% of the dataset).

```

Anaconda Prompt - python
>>> print(dataset.shape)
(150, 5)
>>> print(dataset.head(20))
   sepal-length  sepal-width  petal-length  petal-width  class
0         5.1         3.5         1.4         0.2  Iris-setosa
1         4.9         3.0         1.4         0.2  Iris-setosa
2         4.7         3.2         1.3         0.2  Iris-setosa
3         4.6         3.1         1.5         0.2  Iris-setosa
4         5.0         3.6         1.4         0.2  Iris-setosa
5         5.4         3.9         1.7         0.4  Iris-setosa
6         4.6         3.4         1.4         0.3  Iris-setosa
7         5.0         3.4         1.5         0.2  Iris-setosa
8         4.4         2.9         1.4         0.2  Iris-setosa
9         4.9         3.1         1.5         0.1  Iris-setosa
10        5.4         3.7         1.5         0.2  Iris-setosa
11        4.8         3.4         1.6         0.2  Iris-setosa
12        4.8         3.0         1.4         0.1  Iris-setosa
13        4.3         3.0         1.1         0.1  Iris-setosa
14        5.8         4.0         1.2         0.2  Iris-setosa
15        5.7         4.4         1.5         0.4  Iris-setosa
16        5.4         3.9         1.3         0.4  Iris-setosa
17        5.1         3.5         1.4         0.3  Iris-setosa
18        5.7         3.8         1.7         0.3  Iris-setosa
19        5.1         3.8         1.5         0.3  Iris-setosa
>>> print(dataset.describe())
   sepal-length  sepal-width  petal-length  petal-width
count  150.000000  150.000000  150.000000  150.000000
mean     5.843333    3.054000    3.758667    1.198667
std     0.828066    0.433594    1.764420    0.763161
min     4.300000    2.000000    1.000000    0.100000
25%     5.100000    2.800000    1.600000    0.300000
50%     5.800000    3.000000    4.350000    1.300000
75%     6.400000    3.300000    5.100000    1.800000
max     7.900000    4.400000    6.900000    2.500000
>>> print(dataset.groupby('class').size())
class
Iris-setosa      50
Iris-versicolor  50
Iris-virginica   50
dtype: int64
>>>

```

Figure 3: Information about the iris dataset

1.4.3 Data visualization

We now have a basic idea about the data. We need to extend that with some visualizations.

We are going to look at two types of plots:

1. Univariate plots to better understand each attribute.
2. Multivariate plots to better understand the relationships between attributes.

Univariate plots Given that the input variables are numeric, we can create box and whisker plots of each.

```
dataset.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False)
plt.show()
```

Result can be seen on Figure 4.

We can also create a histogram of each input variable to get an idea of the distribution.

```
dataset.hist()
plt.show()
```

It looks like perhaps two of the input variables have a Gaussian distribution. This is useful to note as we can use algorithms that can exploit this assumption. (Figure 5.)

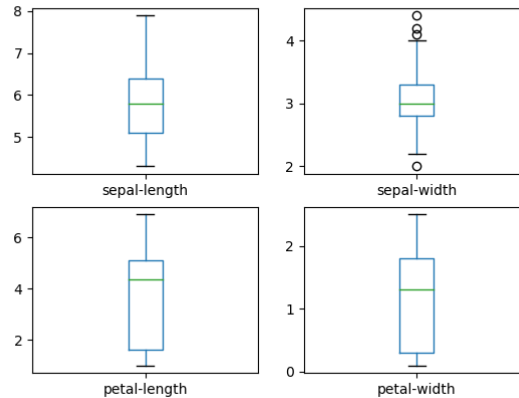


Figure 4: Box and Whisker Plots

Multivariate plots

First, let's look at scatterplots of all pairs of attributes. This can be helpful to spot structured relationships between input variables (Figure 6.). We can note the diagonal grouping of some pairs of attributes. This suggests a high correlation and a predictable relationship.

```
scatter_matrix(dataset)
plt.show()
```

1.4.4 Evaluate Some Algorithms

Then I created some models of the data and estimated their accuracy on unseen data. This step consists of some substeps:

1. Separate out a validation dataset.
2. Set-up the test harness to use 10-fold cross validation.
3. Build 5 different models to predict species from flower measurements
4. Select the best model.

Create a validation dataset

We need to know that the model we created is any good.

Later, I use statistical methods to estimate the accuracy of the models that I created on unseen data. I also want a more concrete estimate of the accuracy of the best model on unseen data by evaluating it on actual unseen data.

That is, I am going to hold back some data that the algorithms will not get to see and I will use this data to get a second and independent idea of how accurate the best model might actually be.

I split the loaded dataset into two, 80% of which I use to train our models and 20% that I hold back as a validation dataset.

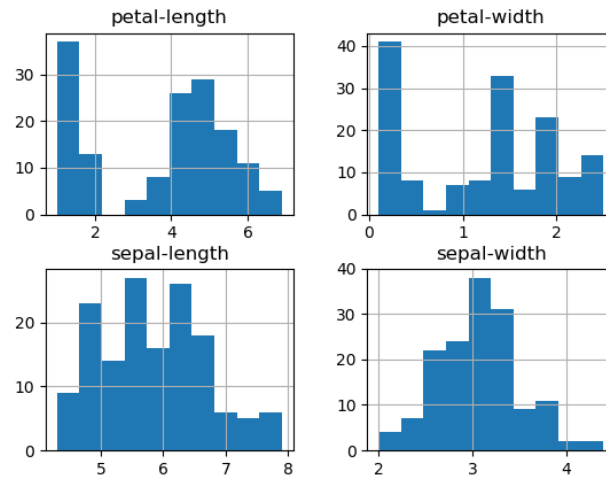


Figure 5: Histogram Plots

```
# Split-out validation dataset
array = dataset.values
X = array[:,0:4]
Y = array[:,4]
validation_size = 0.20
seed = 7
X_train, X_validation, Y_train, Y_validation = model_selection.train_test_split
```

I now have training data in the X_train and Y_train for preparing models and a X_validation and Y_validation sets that I can use later.

Test Harness

I use 10-fold cross validation to estimate accuracy.

This splits our dataset into 10 parts, train on 9 and test on 1 and repeat for all combinations of train-test splits.

```
seed = 7
scoring = 'accuracy'
```

We are using the metric of accuracy to evaluate models. This is a ratio of the number of correctly predicted instances in divided by the total number of instances in the dataset multiplied by 100 to give a percentage (e.g. 95% accurate). I will use the scoring variable when I run, build and evaluate each model next.

Build Models

To get an idea on which algorithms would be good on this problem or what configurations to use, we observe from the plots that some of the classes are partially linearly separable in some dimensions, so we are expecting generally good results.

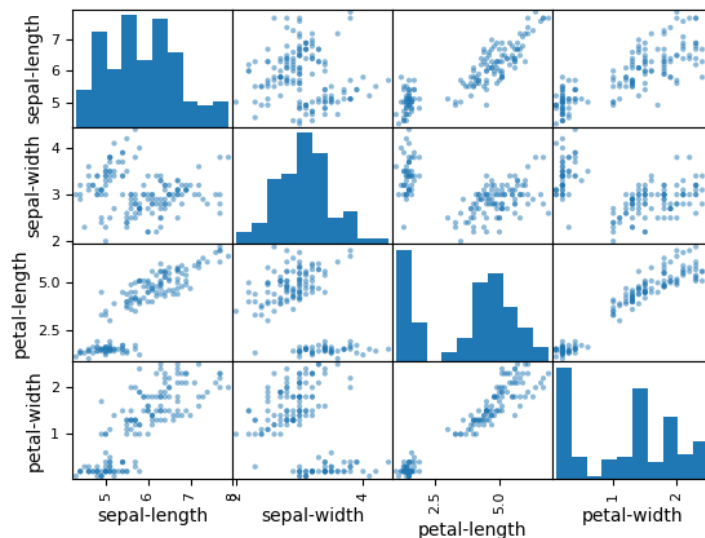


Figure 6: Scatterplot matrix

Lets evaluate 6 different algorithms:

- Logistic Regression (LR)
- Linear Discriminant Analysis (LDA)
- K-Nearest Neighbors (KNN).
- Classification and Regression Trees (CART).
- Gaussian Naive Bayes (NB).
- Support Vector Machines (SVM).

This is a good mixture of simple linear (LR and LDA), nonlinear (KNN, CART, NB and SVM) algorithms. We reset the random number seed before each run to ensure that the evaluation of each algorithm is performed using exactly the same data splits. It ensures the results are directly comparable.

Lets build and evaluate our five models:

```
# Spot Check Algorithms
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
# evaluate each model in turn
results = []
```

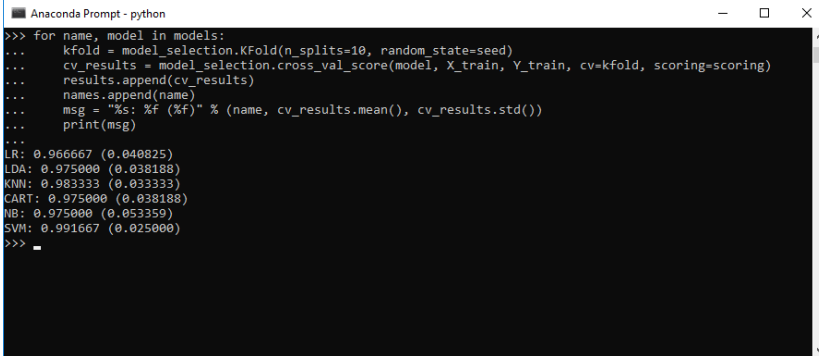
```

names = []
for name, model in models:
    kfold = model_selection.KFold(n_splits=10, random_state=seed)
    cv_results = model_selection.cross_val_score(model, X_train, Y_train, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)

```

Select best model

I now have 6 models and accuracy estimations for each. The result I get after running the commands from above is shown on Figure 7.



```

Anaconda Prompt - python
>>> for name, model in models:
...     kfold = model_selection.KFold(n_splits=10, random_state=seed)
...     cv_results = model_selection.cross_val_score(model, X_train, Y_train, cv=kfold, scoring=scoring)
...     results.append(cv_results)
...     names.append(name)
...     msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
...     print(msg)
...
LR: 0.966667 (0.040825)
LDA: 0.975000 (0.038188)
KNN: 0.983333 (0.033333)
CART: 0.975000 (0.038188)
NB: 0.975000 (0.053359)
SVM: 0.991667 (0.025000)
>>>

```

Figure 7: Result of evaluation the algorithms

From this we see that it looks like KNN has the largest estimated accuracy score.

I can also create a plot of the model evaluation results and compare the spread and the mean accuracy of each model. There is a population of accuracy measures for each algorithm because each algorithm was evaluated 10 times (10 fold cross validation).

```

fig = plt.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()

```

We can see on Figure 8. that the box and whisker plots are squashed at the top of the range, with many samples achieving 100% accuracy.

1.4.5 Make Predictions

The KNN algorithm was the most accurate model that we tested. Now I want to get an idea of the accuracy of the model on our validation set.

This will give us an independent final check on the accuracy of the best model. It is valuable to keep a validation set just in case you made a slip during

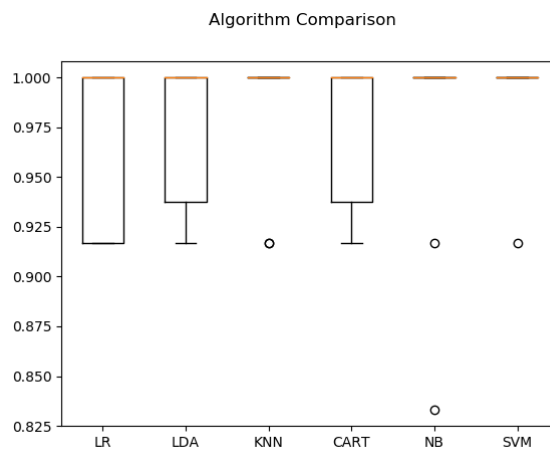


Figure 8: Compare Algorithm Accuracy

training, such as overfitting to the training set or a data leak. Both will result in an overly optimistic result.

I run the KNN model directly on the validation set and summarize the results as a final accuracy score, a confusion matrix and a classification report.

```
knn = KNeighborsClassifier()
knn.fit(X_train, Y_train)
predictions = knn.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
print(confusion_matrix(Y_validation, predictions))
print(classification_report(Y_validation, predictions))
```

We can see on Figure 9. that the accuracy is 0.9 or 90%. The confusion matrix provides an indication of the three errors made. Finally, the classification report provides a breakdown of each class by precision, recall, f1-score and support showing excellent results (granted the validation dataset was small).

```
Anaconda Prompt - python
>>> knn = KNeighborsClassifier()
>>> knn.fit(X_train, Y_train)
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                    weights='uniform')
>>> predictions = knn.predict(X_validation)
>>> print(accuracy_score(Y_validation, predictions))
0.9
>>> print(confusion_matrix(Y_validation, predictions))
[[ 7  0  0]
 [ 0 11  1]
 [ 0  2  9]]
>>> print(classification_report(Y_validation, predictions))
              precision    recall  f1-score   support

 Iris-setosa              1.00      1.00      1.00         7
 Iris-versicolor          0.85      0.92      0.88        12
 Iris-virginica           0.90      0.82      0.86        11

 avg / total              0.90      0.90      0.90        30

>>>
```

Figure 9: Prediction Accuracy Result

2 Proposed problem

2.1 Specification

The tutorial steps from the previous section gave an insight on how the project will look like. First of all we choose a dataset. I chose the dataset of **RSNA Bone Age** [4]. This dataset consists of 12611 X-rays of hands of children. These images have a number associated which is the age in months of the child.

The purpose for this dataset is to be able to predict the age of the child from the X-ray image of the hand.

2.1.1 The dataset

boneage-training-dataset.csv

This csv describes the dataset:

1. id - The number of the image, the image in the images folder is named after this id.png
2. boneage - The age of the child in months
3. male - A boolean value which is true if the child is a boy and false in case it is a girl

boneage-training-dataset folder

Has 12611 .png images of the hand X-rays. The name of the file corresponds to the id in the dataset for which the age is specified.

boneage-test-dataset.csv

This dataset was released for a challenge, thus a test dataset was provided with 200 cases in which the age was not specified, only the gender.

boneage-test-dataset folder

Contains the images corresponding to the test dataset.

For this project, to be able to measure prediction, we will split the provided dataset into training and validation sets. The provided test dataset doesn't specify the correct age, thus we will discard it this time.

2.1.2 Project steps

The project to be developed follows similar steps as the previous tutorial.

1. First we need a summary and a statistical overview of the dataset.
2. Visualize the data to have a representation of how the dataset looks like.
3. Make some analytical decisions and choose learning algorithms which work best on images.
4. split the dataset in 80% training data and 20% test data randomly.

5. Measure accuracy of learning algorithms on the dataset.
6. Run the most accurate learning algorithm on the training data.
7. Run predictions to measure the preciseness of the resulted machine learning process.

2.2 Implementation

We need to split the implementation of this project into three phases:

1. Handle the dataset and process the images
2. Run test algorithms on a chunk of the dataset
3. Run the best performing algorithm on the entire dataset

Source files can be found on [5].

2.2.1 The images

We have 12,611 images in total, with varying sizes, some as big as 2040 x 2570. First of all we need to bring all images to the same size, second, this size should be reasonable. 2,000x2,500 pixels would lead us to 5,000,000 values per image, and with 12,611 images this would be an unreasonably large number of values to process. Thus, with a script in file *resize.py* we create a new folder which contains all the images in a 100x100 size format. Upon examining the original images we can observe that some images have excess borders and some bones are rotated within the image. I did not recur to image processing algorithms to fix this, as that would require way too much unrelated work.

As a result, the images got shrinked and stretched and I saved them in a folder called *boneage-resized-dataset*.

2.2.2 Testing algorithms

In order to see the functionality of the project, in file *prediction.py* I did some testing on a chunk of the dataset, let's say the first 1000 records in the dataset.

To be able to work with the image data and to run regression algorithms on it, first we put all images in an array, we convert it to a numpy array and then we reshape them each to be an array of numbers, not a matrix of 100x100. After this was done, I chose 4 supervised learning regression algorithms to test their efficiency on the dataset. To perform this, I called the method *cross_val_score* on the chunk of data and the target values with *cv=5*. This means that the algorithm is called 5 times, learning every time from part of the data, then computing the negative mean absolute error on the predictions of the other part. The algorithms I chose were:

- Regression based on k-nearest neighbors: **KNeighborsRegressor()**
- Multi-layer Perceptron regressor: **MLPRegressor()**
- Gaussian process regression: **GaussianProcessRegressor()**
- Epsilon-Support Vector Regression: **SVR()**

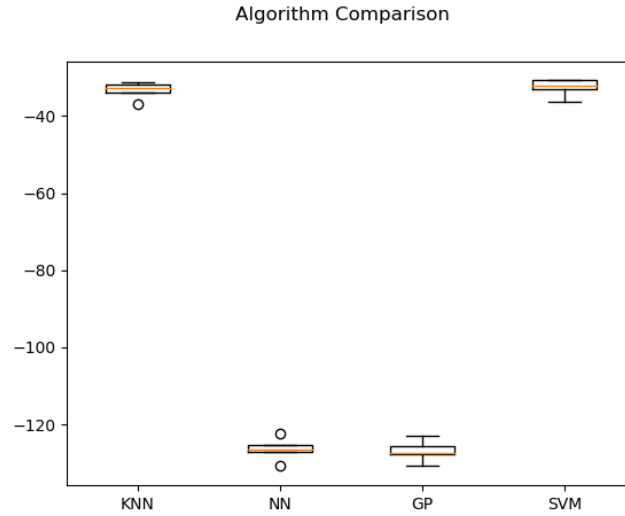


Figure 10: Algorithms Neg. Mean Absolute Error scoring

Even though, as Figure 10. presents, both the k-nearest neighbors algorithm and the support vector one performed well, I chose to work with the Epsilon-Support Vector Regression, as decided upon the values of the mean of the negative mean absolute errors, as listed below:

- KNN: -33.284800 (2.042105)
- NN: -126.380471 (2.735587)
- GP: -126.863000 (2.627972)
- SVM: -32.643243 (2.005659)

Then numbers in the parentheses are the standard deviation values.

After choosing the algorithm, I tried out evaluating it on a pipelined version of the data. The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. I used as scaler `StandardScaler()`, which standardizes features by removing the mean and scaling to unit variance.

Cross validating SVR on this pipeline with `cv=10` gave a mean of *-35.61575216284298* so I decided not to go with the pipelined version.

To visualize the predictions, I ran the learning algorithm on the entire chunk of data except the last four. Then, I told the algorithm to predict the bone age of the last four images. The results can be seen on Figure 11.

For SVR there can be multiple kernel types specified, and I tried out the following ones:

```
svr_rbf = SVR(kernel='rbf', gamma=0.1)
svr_lin = SVR(kernel='linear')
svr_poly = SVR(kernel='poly', degree=2)
```

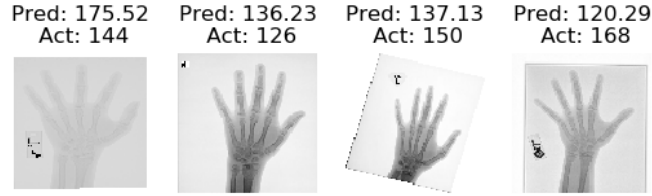



Figure 11: Predicted values with SVR for 4 images

When no values were specified for the argument *kernel*, then RBF was used.

Each one has its own error result and I visualized the difference of the original value and the predicted one on graphs, for all three kernel types I used. (Figure 12.)

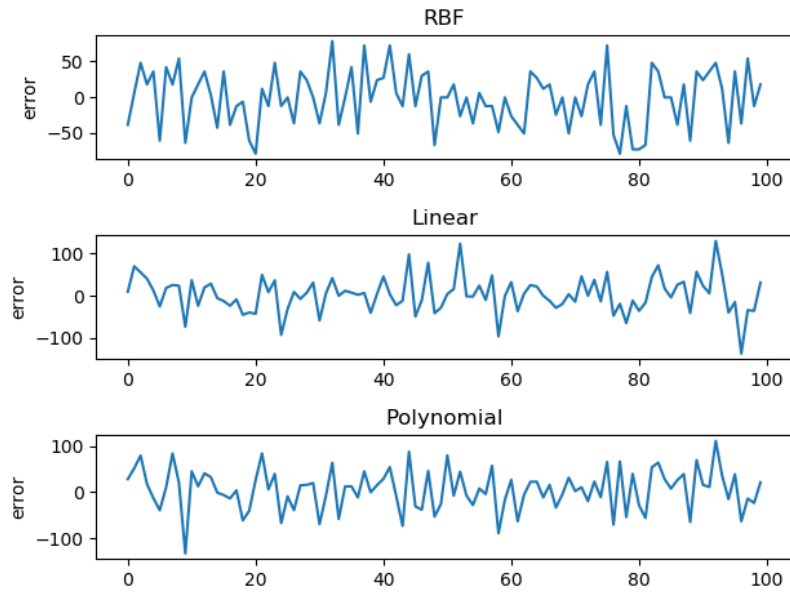


Figure 12: Differences between actual and predicted values for three kernel types of the SVR algorithm

Judging by these graphs we would say the linear kernel gives the most appropriate results, as its line doesn't vary that far from the Ox axis on multiple segments. But its range of differences goes on the y axis between $[-100, 100]$ in contrast to the RBF kernel, which has more varying differences, but between $[-50, 50]$. Cross validating these three kernels with $cv=10$ and displaying the mean of errors confirms that the RBF kernel SVR algorithm gives the smaller

error result:

- RBF: -32.642578 (2.583530)
- Linear: -35.730347 (3.131138)
- Polynomial: -37.911338 (4.356987)

Regarding all this testing, I decided to run the SVR algorithm with the RBF kernel on the entire (resized) dataset.

2.2.3 The Algorithm

SVR RBF Support Vector Machine (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, it is mostly used in classification problems. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiate the two classes very well.[6] (Figure 13.)

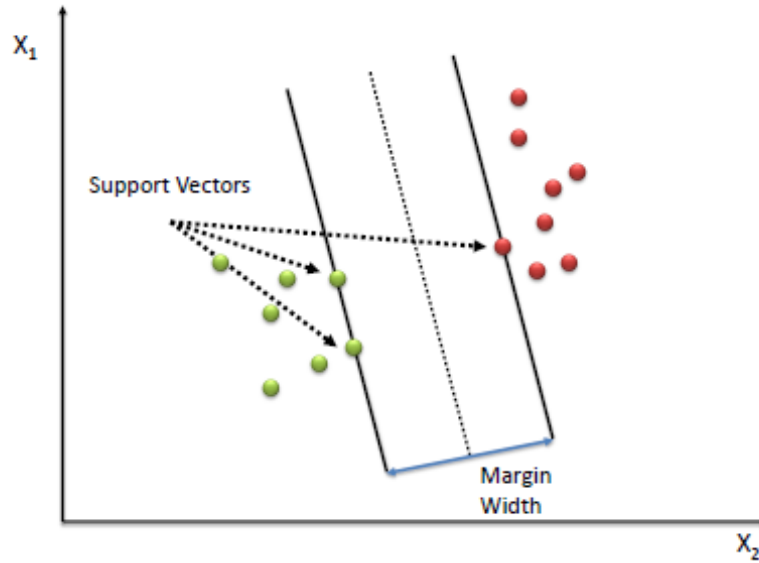


Figure 13: Visual representation of the SVM algorithm

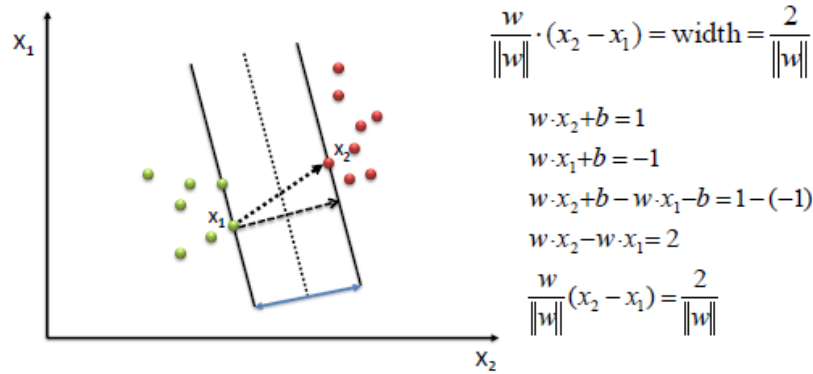
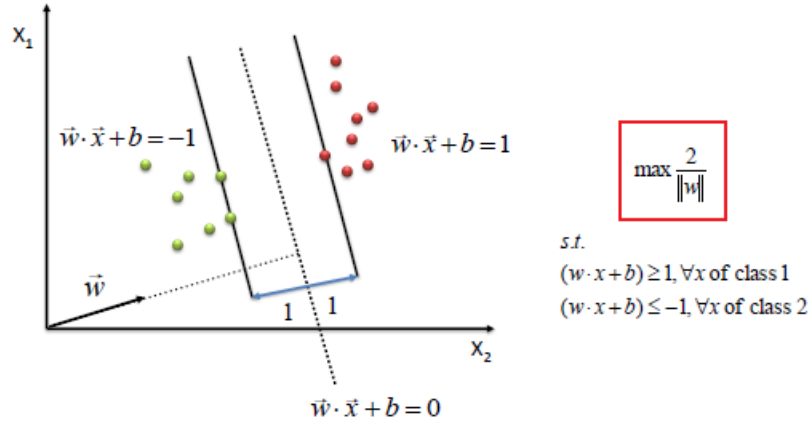
Support Vectors are simply the co-ordinates of individual observation. Support Vector Machine is a frontier which best segregates the two classes (hyper-plane/ line).

Algorithm

1. Define an optimal hyperplane: maximize margin
2. Extend the above definition for non-linearly separable problems: have a penalty term for mis-classifications.

3. Map data to high dimensional space where it is easier to classify with linear decision surfaces: reformulate problem so that data is mapped implicitly to this space.

To define an optimal hyperplane we need to maximize the width of the margin (w). The following figures visualize the process and the mathematical procedure.



We find w and b by solving the following objective function using Quadratic Programming.

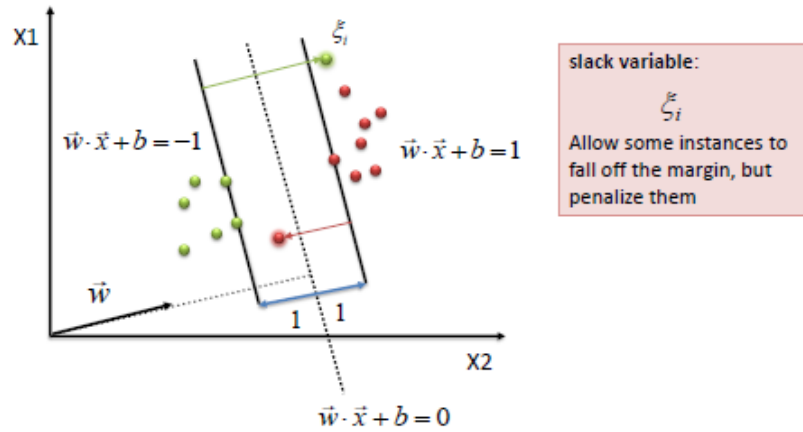
$$\min \frac{1}{2} \|w\|^2$$

such that

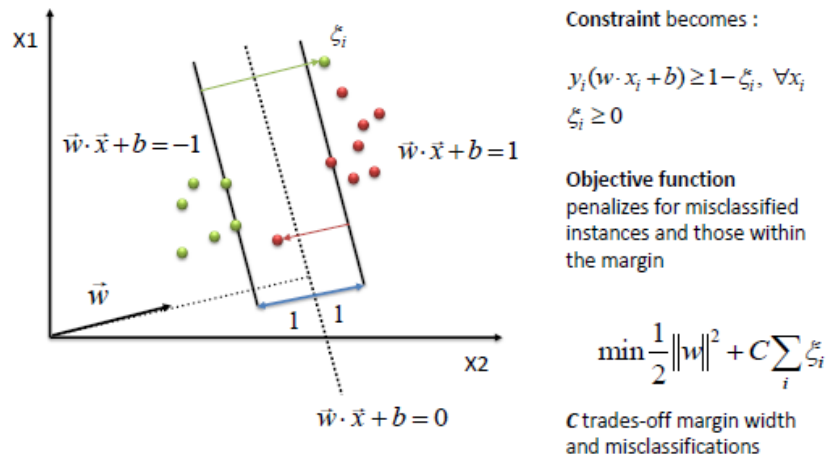
$$y_i(w \cdot x_i + b) \leq 1 \forall x_i$$

The beauty of SVM is that if the data is linearly separable, there is a unique global minimum value. An ideal SVM analysis should produce a hyperplane that completely separates the vectors (cases) into two non-overlapping classes.

However, perfect separation may not be possible, or it may result in a model with so many cases that the model does not classify correctly. In this situation SVM finds the hyperplane that maximizes the margin and minimizes the mis-classifications.

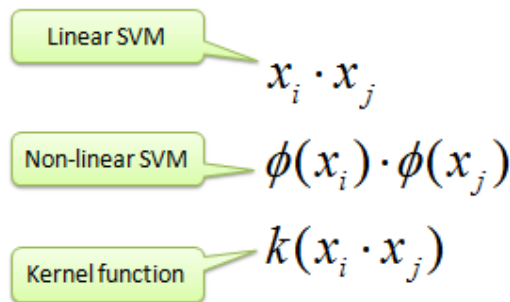
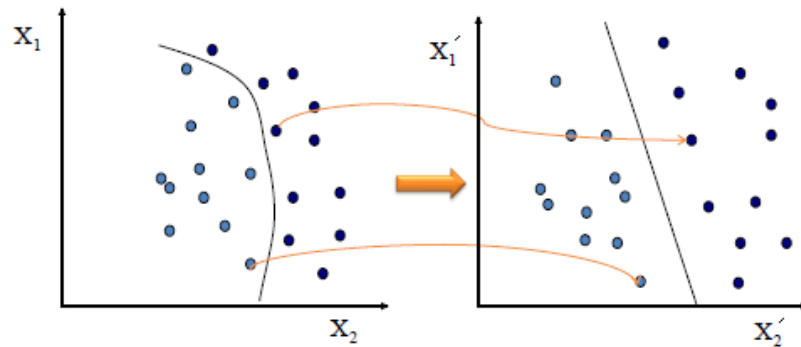


The algorithm tries to maintain the slack variable to zero while maximizing margin. However, it does not minimize the number of mis-classifications (NP-complete problem) but the sum of distances from the margin hyperplanes.



The simplest way to separate two groups of data is with a straight line (1 dimension), flat plane (2 dimensions) or an N-dimensional hyperplane. However, there are situations where a nonlinear region can separate the groups more efficiently. SVM handles this by using a kernel function (nonlinear) to map the data into a different space where a hyperplane (linear) cannot be used to do the separation. It means a non-linear function is learned by a linear learning machine in a high-dimensional feature space while the capacity of the system is controlled by a parameter that does not depend on the dimensionality of the space. This is called kernel trick which means the kernel function transform the

data into a higher dimensional feature space to make it possible to perform the linear separation.



Map data into new space, then take the inner product of the new vectors. The image of the inner product of the data is the inner product of the images of the data. Two kernel functions are shown below.

Polynomial

$$k(x_i, x_j) = (x_i \cdot x_j)^d$$

Gaussian Radial Basis function

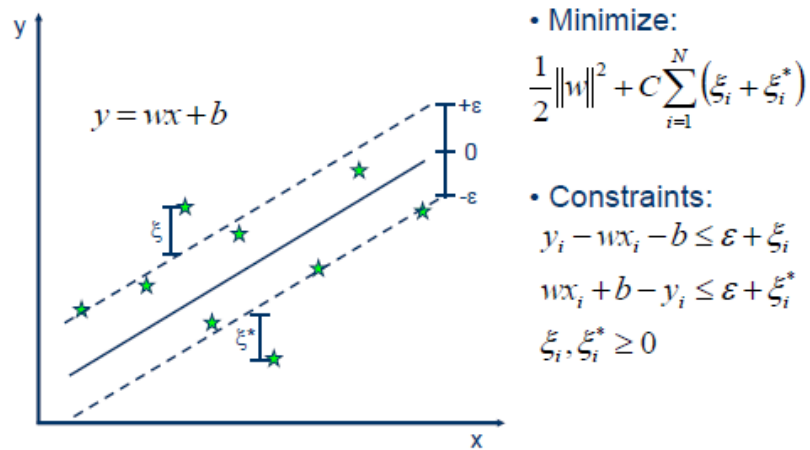
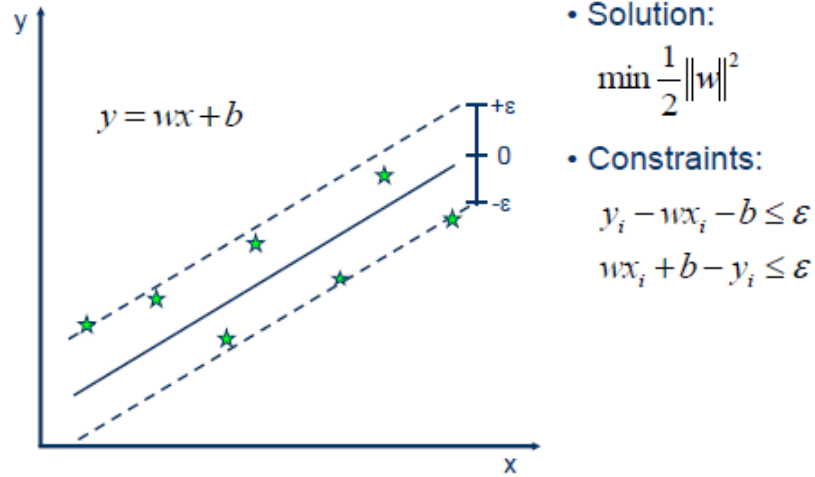
$$k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$$

We use the latter one for this project.

Regression

Support Vector Machine is also used as a regression method, the algorithm we use for this project. It maintains all the main features that characterize the algorithm (maximal margin). The Support Vector Regression (SVR) uses the same principles as the SVM for classification, with only a few minor differences. First of all, because output is a real number it becomes very difficult to predict the information at hand, which has infinite possibilities. In the case of regression, a

margin of tolerance (epsilon) is set in approximation to the SVM which would have already requested from the problem. But besides this fact, there is also a more complicated reason, the algorithm is more complicated therefore to be taken in consideration. However, the main idea is always the same: to minimize error, individualizing the hyperplane which maximizes the margin, keeping in mind that part of the error is tolerated.



The algorithm can be categorized in two main types: **Linear SVR**

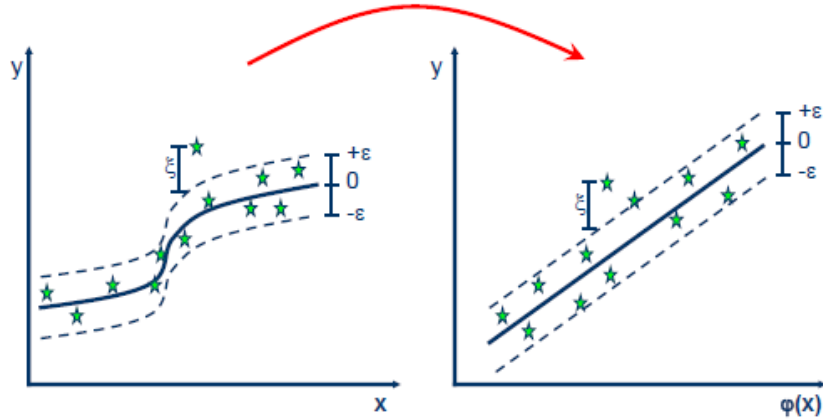
$$y = \sum_{i=1}^N (\alpha_i - \alpha_i^*) \cdot \langle x_i, x \rangle + b$$

Non-linear SVR The kernel functions transform the data into a higher dimensional feature space to make it possible to perform the linear separation.

RBF is a nonlinear kernel.

$$y = \sum_{i=1}^N (\alpha_i - \alpha_i^*) \cdot \langle \phi(x_i), \phi(x) \rangle + b$$

$$y = \sum_{i=1}^N (\alpha_i - \alpha_i^*) \cdot K(x_i, x) + b$$



The syntax of the SVR class constructor within scikit-learn from class *sklearn.svm* looks like the following:

```
SVR(kernel='rbf', degree=3, gamma='auto', coef0=0.0, tol=0.001,
C=1.0, epsilon=0.1, shrinking=True, cache_size=200, verbose=False,
max_iter=-1)
```

The parameters and the methods that can be called upon it are explained on the main scikit-learn website[7].

2.2.4 Results

After reading and reshaping the data both for the entire training dataset as for the testing dataset that we generated, we instantiate the Support Vector Regression with the following parameters:

```
svr_rbf = SVR(kernel='rbf', gamma=0.1)
```

We call the *fit* method on the training dataset and the *predict* method on the testing data and store the predictions in an array.

```
y_rbf = svr_rbf.fit(data, target).predict(data_test)
```

We plot the errors by id, being the difference between the actual value and the predicted value. The results can be seen on Figure 14.

To show an accuracy result, we call cross-validation on the entire dataset with a *cv* equal to 10. The resulting mean and the standard deviation of the 10 runs is:

```
RBF: -33.037890 (0.600197)
```

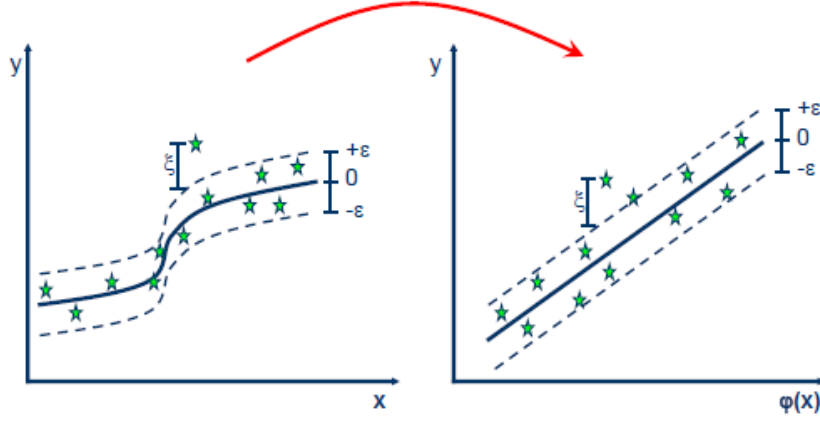


Figure 14: Plotted errors after training on the entire training dataset, and predicting the entire testing dataset

2.3 Conclusions

Training and predicting on this dataset would have required first image processing on the entire dataset, such as recognizing the hand bone structure and rotating and scaling each image such that all hands fit within the same sized region. The images were high quality, hence processing and regarding all this and letting the algorithm work on greater sizes for more time would probably give better results.

The other aspect of this project was that the dataset was not evenly distributed and because we had to predict a number between 0 and 228 (the maximum number found within this dataset) we chose regression. Thus, the algorithm also gave decimal numbers as a result.

Considering all these, I would say the results are not bad, we can even see an improvement between training on a chunk of 1000 and training on the entire dataset.

Overall, further developments could be made, but the algorithm performed well, as we have errors within a bounded interval.

2.4 Further developments

As mentioned before, applying image processing algorithms first would help out the learning a lot.

Also, running on full-sized images would consider a higher resolution and more correlations to train on.

Another further research on the dataset would be to predict age in years, not months. After converting months to years throughout the dataset, we would end up with 19 classes, not 228, thus classification algorithms could be considered.

References

- [1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [2] <https://en.wikipedia.org/wiki/Scikit-learn>.
- [3] <https://machinelearningmastery.com/machine-learning-in-python-step-by-step>.
- [4] <https://www.kaggle.com/kmader/rsna-bone-age>.
- [5] <https://github.com/icefairy2/boneage-IS>.
- [6] http://www.saedsayad.com/support_vector_machine_reg.htm.
- [7] <http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html>.