王萧诚 2019211318 自动化系 控制工程

Ⅰ. Decision Tree and Embedded Learning

1. Study and run http://scikit-learn.org/stable/modules/tree.html

a) Understand the meanings of parameters in decision tree algorithm through the instruction **help(tree.DecisionTreeClassifier)**

```
def __init__(self,
             criterion="gini",
             splitter="best",
             max_depth=None,
             min_samples_split=2,
             min_samples_leaf=1,
             min_weight_fraction_leaf=0.,
             max_features=None,
             random_state=None,
             max_leaf_nodes=None,
             min_impurity_decrease=0.,
             min_impurity_split=None,
             class_weight=None,
             presort=False):
    super().__init__(
        criterion=criterion,
        splitter=splitter,
        max_depth=max_depth,
        min_samples_split=min_samples_split,
        min_samples_leaf=min_samples_leaf,
        min_weight_fraction_leaf=min_weight_fraction_leaf,
        max_features=max_features,
        max_leaf_nodes=max_leaf_nodes,
        class_weight=class_weight,
        random_state=random_state,
        min_impurity_decrease=min_impurity_decrease,
        min_impurity_split=min_impurity_split,
        presort=presort)
```

Fig.1 parameters of DecisionTreeClassifier()

From Fig.1 we can see the **init** function of class DecisionTreeClassifier in tree.py. There are several important parameters like **criterion, max_depth, max_leaf**. We can see the explanation of each parameter in the **appendix page**.

b) Import iris dataset, write a program that can open the dataset and divide it into training set and test set. Use gini criterion and entropy criterion to train the decision tree and visualize the model respectively. How many leaf nodes are there in the default model? Change the max_leaf_nodes parameter from 9 to 3, check the changes in the structure of model and explain it. Change the max_depth parameter, discuss the impact of depth on the model.

With **graphviz** and **pydotplus** libraries we can visualize the trained decision trees with **gini** or **entropy** criterion.

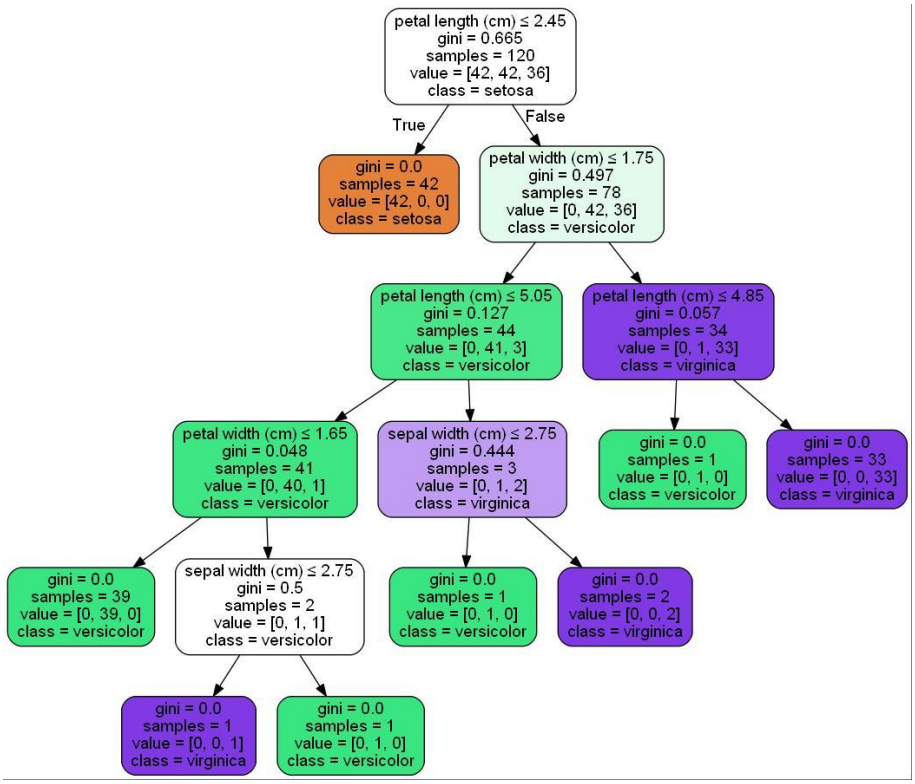

Fig.2 Decision Tree trained with gini criterion

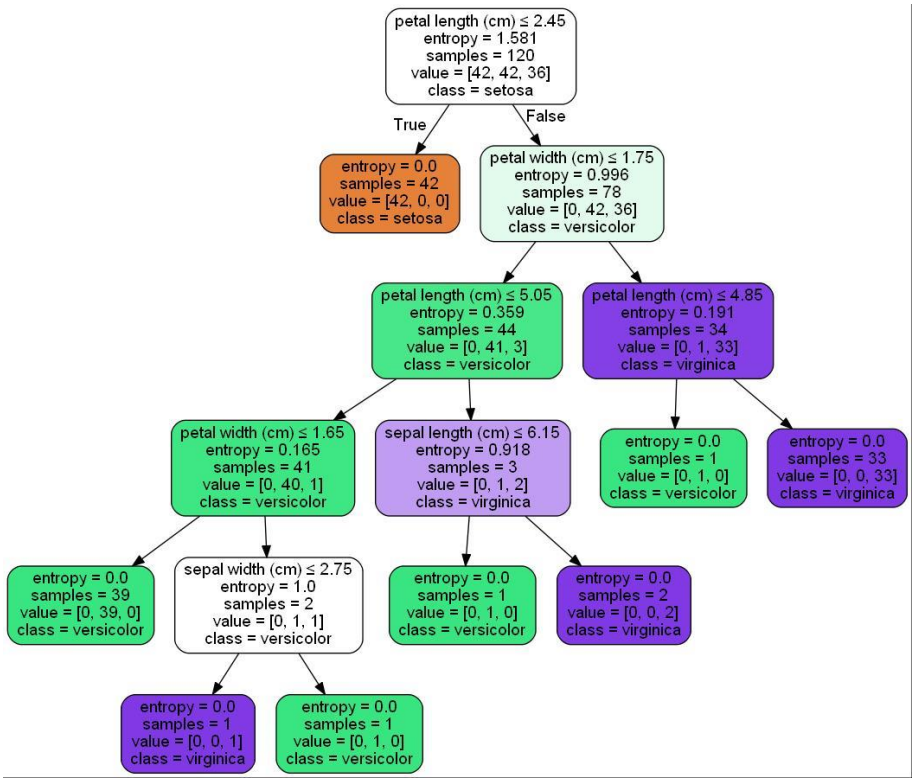

Fig.3 Decision Tree trained with entropy criterion

It can be seen that even trained with different criteria, the results of decision trees are rathe the same. Because both criteria are indicators of the impurity and it is easy to find out that both criteria are 0 at the leaf node. The only difference is the computing formula.

c) Study Multi-output Decision Tree Regression from http://scikit-learn.org/stable/auto_examples/tree/plot_tree_regression_multioutput.html The multi-output decision tree is realized by another function of decision tree model, tree.**DecisionTreeRegressor**(), not tree.DecisionTreeClassifier. Most parameters of these two functions are the same. Parameters of tree.DecisionTreeRegressor are shown below.

```
def __init__(self,
             criterion="mse",
             splitter="best",
             max_depth=None,
             min_samples_split=2,
             min_samples_leaf=1,
             min_weight_fraction_leaf=0.,
             max_features=None,
             random_state=None,
             max_leaf_nodes=None,
             min_impurity_decrease=0.,
             min_impurity_split=None,
             presort=False):
```

Fig.4 Parameters of tree.DecisionTreeRegressor()

d) Use the above decision tree model to solve the face completion problem. Change the parameters of the model and discuss the resulting impact. Reference: http://scikit-learn.org/stable/auto_examples/plot_multioutput_face_completion.html#example-plot-multioutput-face-completion-py

2. Use Random Forests to solve the face completion problem. Give the visualized comparison of it with general decision tree method. Reference: http://scikit-learn.org/stable/auto_examples/ensemble/plot_random_forest_regression_multioutput.html#sphx-glr-auto-examples-ensemble-plot-random-forest-regression-multioutput-py

Here we put the results of **multi-output decision tree** and **random forest** together. We changed the **max_depth** from 5 to 10, 20, 30, 40. The visualized results of both methods as well as that of other methods are shown in following figures. Note that we did not change the parameters of other models. These models are just of general comparison.

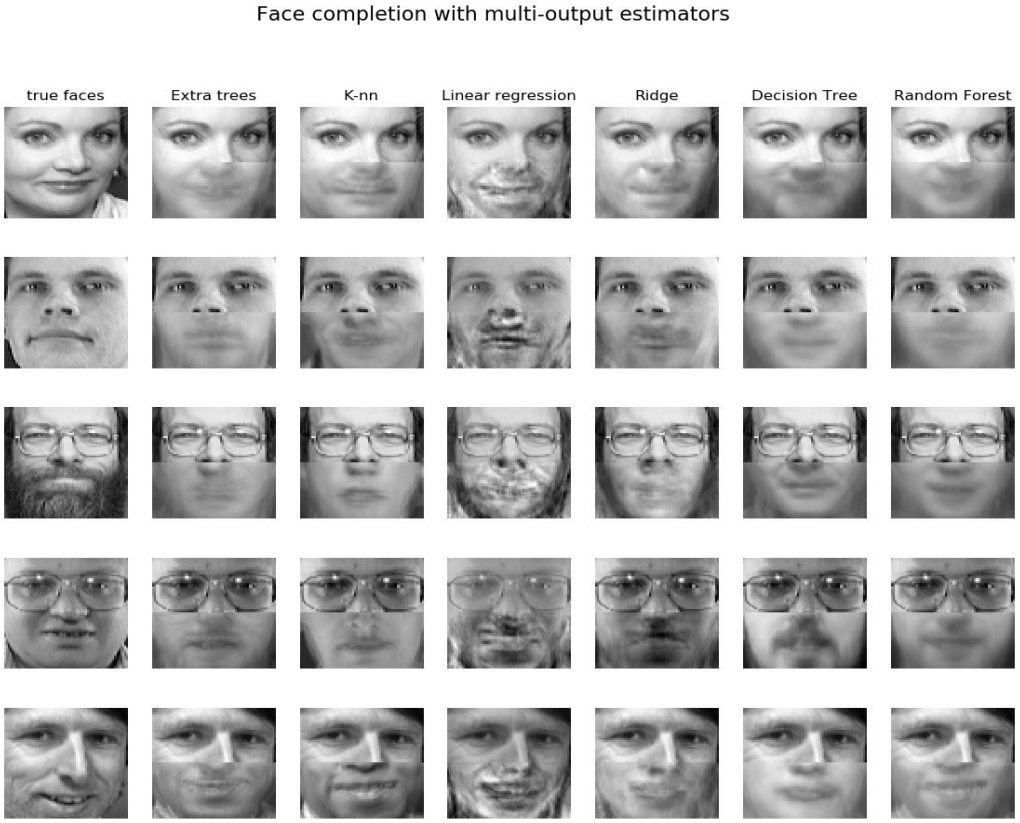

Fig.5 max_depth = 5

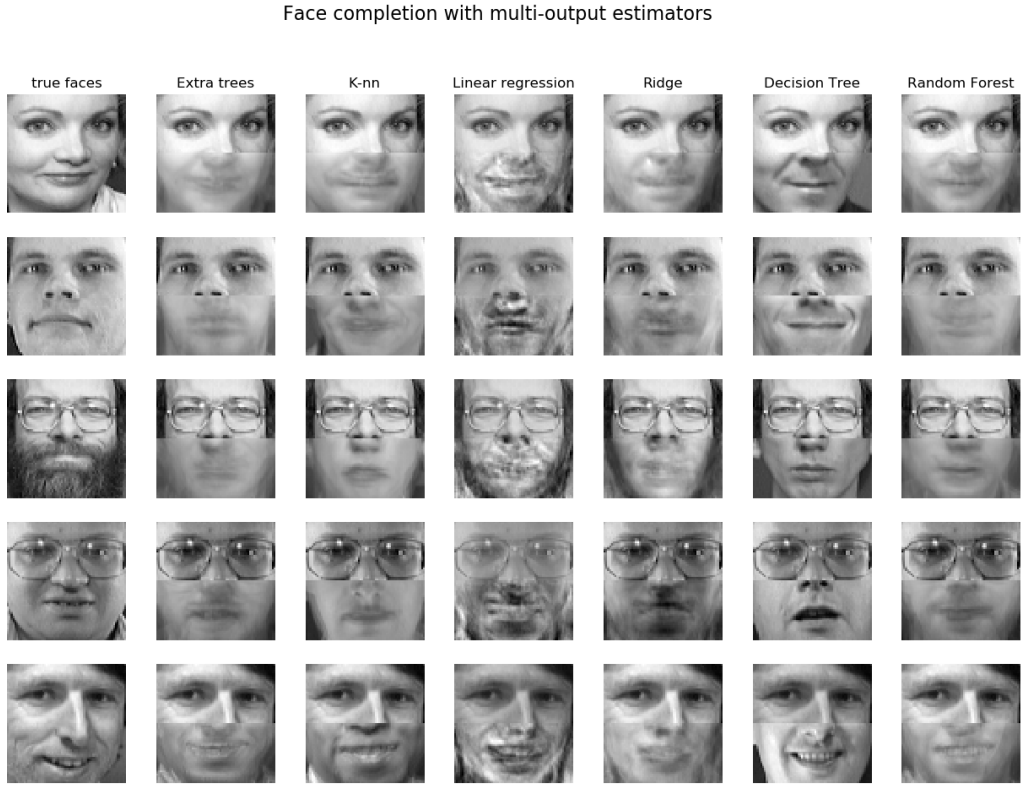

Fig.6 max_depth = 10

Fig.7 max_depth = 20

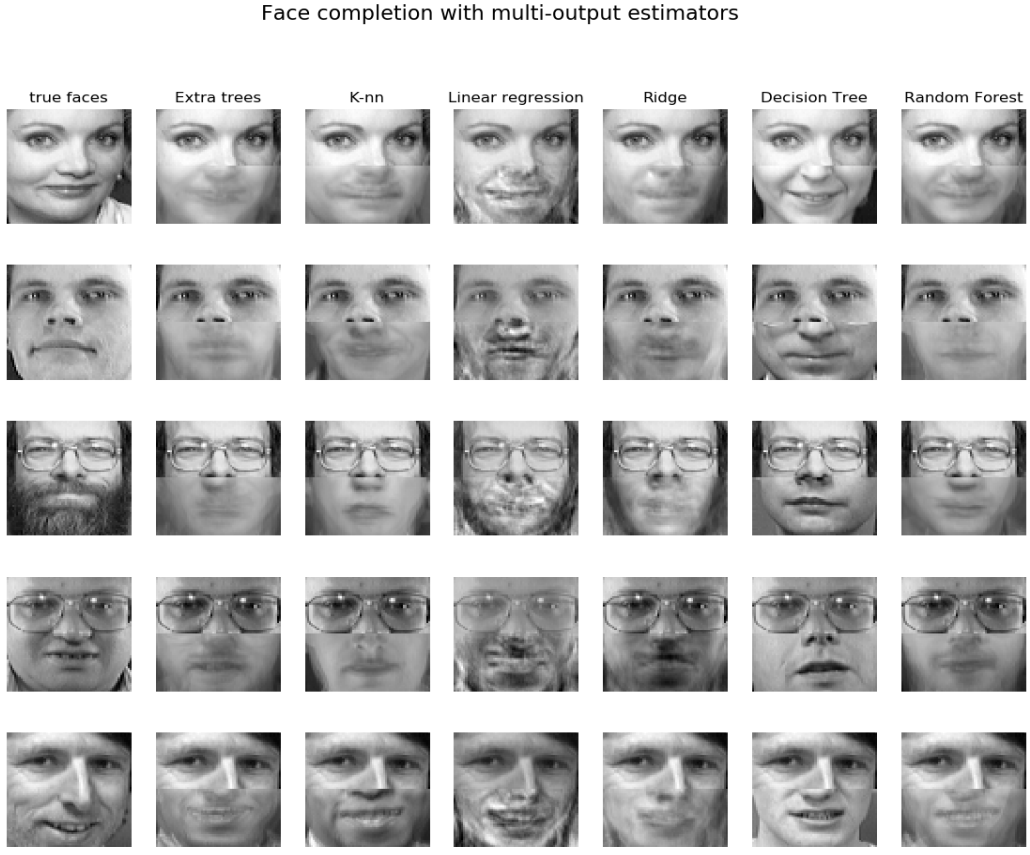

Fig.8 max_depth = 30

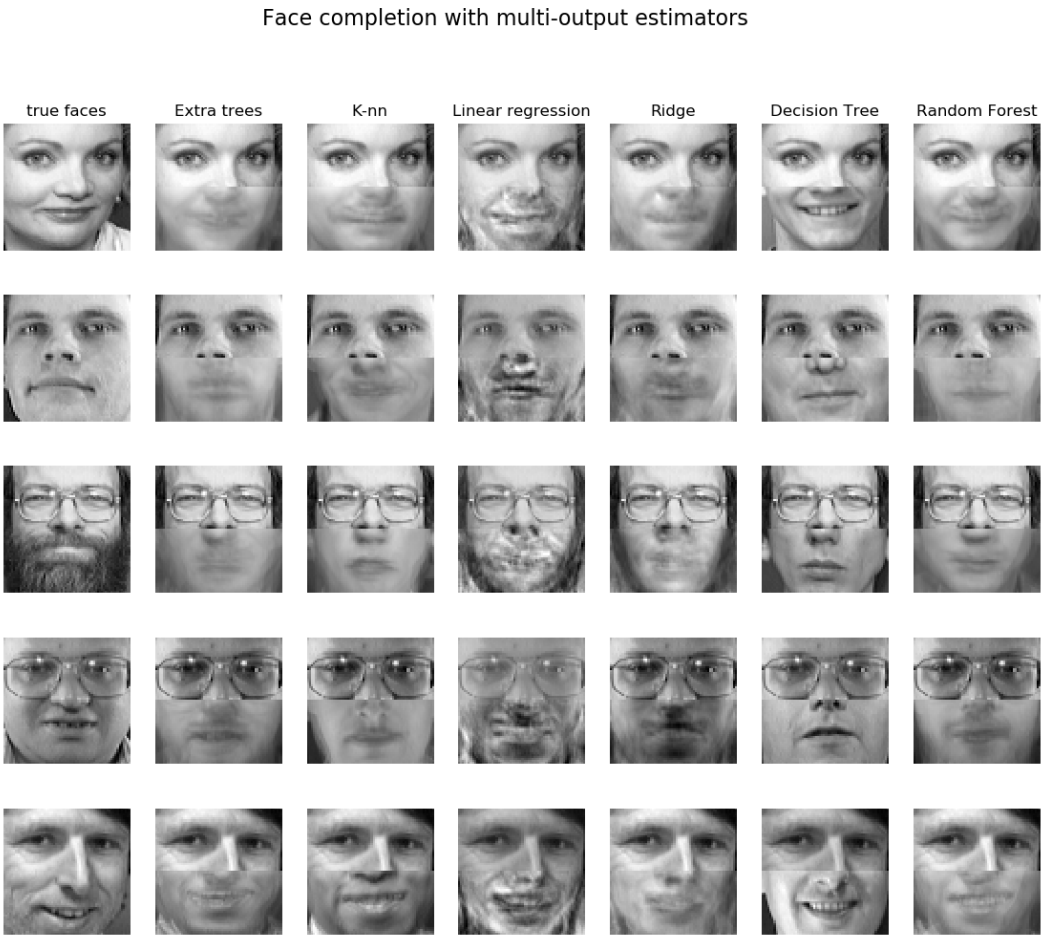

Fig.9 max_depth = 40

From Fig.5-9 we can see that as the max_depth increases, the performance of random forest is rather **stable** while the performance of decision tree **changes a lot**. As the max_depth increases, the completion of picture generated by decision tree becomes **clearer** and reaches the best performance at around **max_depth = 30** and it seems overfit at max_depth = 40, but still, it is **worse** than random forest as we see it.

II . SVM Classification and SVR Regression

1. SVR Regression

a) Understand the impact of different kernel functions on the model from http://scikit-learn.org/stable/auto_examples/svm/plot_svm_regression.html#sphx-glr-auto-examples-svm-plot-svm-regression-py

Generally, there are 3 types of kernel function: linear, polynomial, RBF(Gauss).

Linear kernel is applied in situation where data are **linear separable**. Since it has less parameters, its training procedure is rather **fast**.

Polynomial kernel has most parameters and as the order rises, the model becomes more **complex**.

RBF kernel is the **most widely applied** kernel and can be applied in **linear inseparable** situations. It has **less parameters** than polynomial kernel. In fact, linear kernel is a **special**

case of RBF kernel, therefore, RBF kernel can also deal with linear separable situations. Generally, when we first meet a dataset, if the **feature dimension** is very **large** and almost has the same size of the dataset, then it is usually **linear separable**, so we first try **linear** kernel. If the dataset is very large but features are very few, then we can manually **add some features** and then use linear kernel. If the dataset is rather **small** and **linear inseparable**, then we should consider **RBF** kernel. **Polynomial kernel is less considered**. There is another kernel, **sigmoid kernel** is very few considered when we use SVM since in this case it is a **neural network** model.

b) Write a program to implement SVR regression on Boston house price and evaluate it. Here we use SVR with 3 different kernels and compare the respective results.

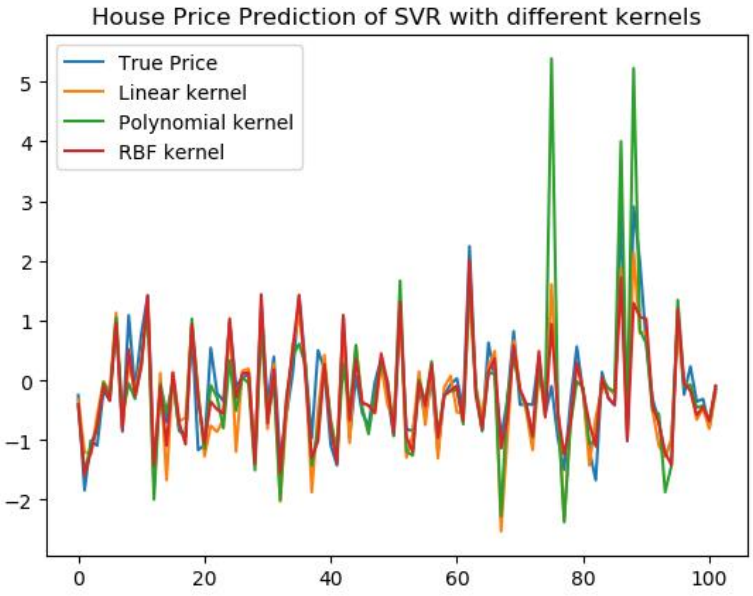

Fig.10 Prediction of House Price

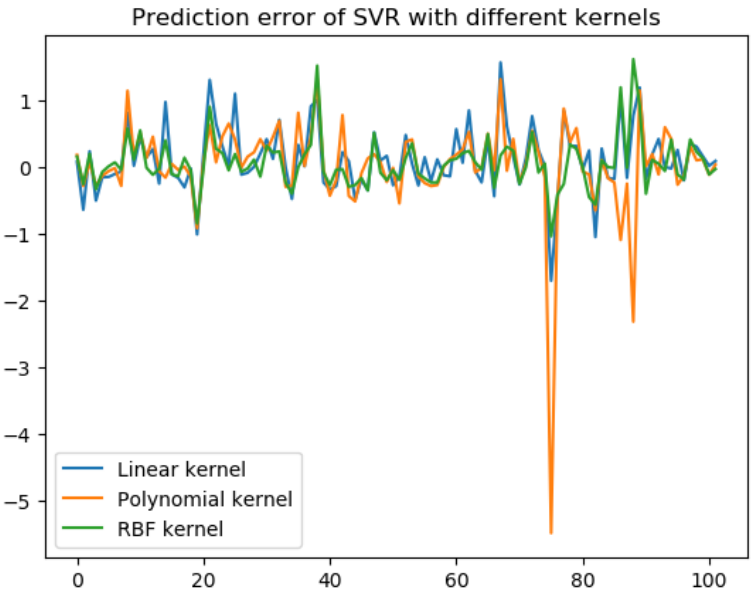

Fig.11 Prediction error of different kernels

```
_________
Linear Kernel
Score:  0.6907030841585
R2:  0.6907030841585
MSE: 22.156320166849437
MAE: 3.3579016195988634

_________
Polynomial Kernel
Score:  0.354252758193746
R2:  0.354252758193746
MSE: 46.25776043512556
MAE: 3.591475200825326

_________
RBF Kernel
Score:  0.8173380557351699
R2:  0.8173380557351698
MSE: 13.084891287777145
MAE: 2.389716279398295
```

Fig.11 Evaluation of different kernels

From Fig.10-12 we can see that in the Boston house price dataset, **RBF** kernel shows the best performance over other 2 kernels. As we have analyzed before, in this dataset, features are not too many and the dataset is not too large and this is just the case where RBF is well suited.

2. SVM Classification

a) Learn the svm fuction in scikit-learn.

Reference: http://scikit-learn.org/stable/modules/svm.html#svm

Since we have learned the SVM method in last experiment, we do not explain too much of the basic theory here.

b) Train SVM on hand-written digits pictures with different kernel functions and evaluate the respective results.

Still, we train SVM with linear kernel, polynomial kernel and RBF kernel. Here are the results and evaluations.

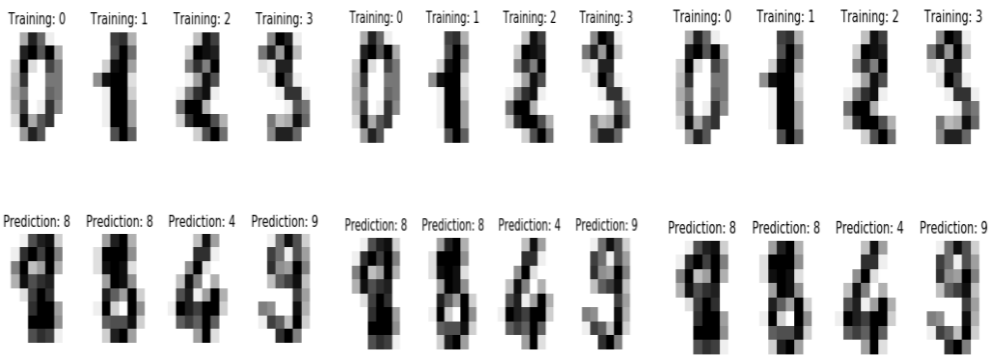

Fig.12 Demonstration of prediction with linear, polynomial, RBF kernel

	precision	recall	f1-score	support
0	0.97	0.99	0.98	88
1	0.94	0.90	0.92	91
2	1.00	0.99	0.99	86
3	0.97	0.86	0.91	91
4	0.99	0.95	0.97	92
5	0.90	0.97	0.93	91
6	0.98	0.99	0.98	91
7	0.97	0.96	0.96	89
8	0.88	0.92	0.90	88
9	0.87	0.93	0.90	92
accuracy			0.94	899
macro avg	0.95	0.94	0.94	899
weighted avg	0.95	0.94	0.94	899

Confusion matrix:

```
[[87  0  0  0  0  0  1  0  0  0]
 [ 0 82  0  0  0  0  0  0  3  6]
 [ 1  0 85  0  0  0  0  0  0  0]
 [ 0  0  0 78  0  4  0  1  8  0]
 [ 1  0  0  0 87  0  0  0  0  4]
 [ 0  0  0  0  0 88  1  0  0  2]
 [ 0  1  0  0  0  0 90  0  0  0]
 [ 0  1  0  0  1  2  0 85  0  0]
 [ 0  3  0  1  0  1  0  1 81  1]
 [ 1  0  0  1  0  3  0  1  0 86]]
```

	precision	recall	f1-score	support
0	0.99	0.98	0.98	88
1	0.99	0.92	0.95	91
2	1.00	0.97	0.98	86
3	0.94	0.89	0.92	91
4	0.99	0.96	0.97	92
5	0.93	0.97	0.95	91
6	0.98	0.99	0.98	91
7	0.97	0.98	0.97	89
8	0.91	0.97	0.94	88
9	0.88	0.95	0.91	92
accuracy			0.96	899
macro avg	0.96	0.96	0.96	899
weighted avg	0.96	0.96	0.96	899

Confusion matrix:

```
[[86  0  0  0  1  0  1  0  0  0]
 [ 0 84  0  0  0  0  0  0  2  5]
 [ 1  0 83  2  0  0  0  0  0  0]
 [ 0  0  0 81  0  3  0  1  5  1]
 [ 0  0  0  0 88  0  0  0  0  4]
 [ 0  0  0  0  0 88  1  0  0  2]
 [ 0  1  0  0  0  0 90  0  0  0]
 [ 0  0  0  0  0  1  0 87  1  0]
 [ 0  0  0  1  0  1  0  1 85  0]
 [ 0  0  0  2  0  2  0  1  0 87]]
```

	precision	recall	f1-score	support
0	1.00	0.99	0.99	88
1	0.99	0.97	0.98	91
2	0.99	0.99	0.99	86
3	0.98	0.87	0.92	91
4	0.99	0.96	0.97	92
5	0.95	0.97	0.96	91
6	0.99	0.99	0.99	91
7	0.96	0.99	0.97	89
8	0.94	1.00	0.97	88
9	0.93	0.98	0.95	92
accuracy			0.97	899
macro avg	0.97	0.97	0.97	899
weighted avg	0.97	0.97	0.97	899

Confusion matrix:

```
[[87  0  0  0  1  0  0  0  0  0]
 [ 0 88  1  0  0  0  0  0  1  1]
 [ 0  0 85  1  0  0  0  0  0  0]
 [ 0  0  0 79  0  3  0  4  5  0]
 [ 0  0  0  0 88  0  0  0  0  4]
 [ 0  0  0  0  0 88  1  0  0  2]
 [ 0  1  0  0  0  0 90  0  0  0]
 [ 0  0  0  0  0  1  0 88  0  0]
 [ 0  0  0  0  0  0  0  0 88  0]
 [ 0  0  0  1  0  1  0  0  0 90]]
```

Fig.13 Evaluation of linear, polynomial, RBF kernel

From Fig.12 and Fig.13 we can find that SVM trained with all these three kernels shows good performance with f1-score: 0.94/0.96/0.97 for linear/polynomial/RBF respectively. Since this is not a large dataset and as we have analyzed above, RBF shows the best performance.

III . Conclusion

Through this experiment I gained deeper acknowledgement of Decision Tree and SVM method, especially how each parameter affects the overall model. Meanwhile I practiced Python coding skills before this experiment, so I feel a little bit easier than last experiment. There are still some techniques which I do not practiced in this experiment such as pruning for decision tree and soft-margin SVM, etc. It seems that even such basic and rather simple model has these deeper theory behind and I think we should not just know how to code and implement these method but also take grasp of those theory behind.

```
criterion : string, optional (default="gini")
    The function to measure the quality of a split. Supported criteria are
    "gini" for the Gini impurity and "entropy" for the information gain.

splitter : string, optional (default="best")
    The strategy used to choose the split at each node. Supported
    strategies are "best" to choose the best split and "random" to choose
    the best random split.

max_depth : int or None, optional (default=None)
    The maximum depth of the tree. If None, then nodes are expanded until
    all leaves are pure or until all leaves contain less than
    min_samples_split samples.

min_samples_split : int, float, optional (default=2)
    The minimum number of samples required to split an internal node:

    - If int, then consider `min_samples_split` as the minimum number.
    - If float, then `min_samples_split` is a fraction and
      `ceil(min_samples_split * n_samples)` are the minimum
      number of samples for each split.

    .. versionchanged:: 0.18
       Added float values for fractions.

min_samples_leaf : int, float, optional (default=1)
    The minimum number of samples required to be at a leaf node.
    A split point at any depth will only be considered if it leaves at
    least ``min_samples_leaf`` training samples in each of the left and
    right branches.  This may have the effect of smoothing the model,
    especially in regression.

    - If int, then consider `min_samples_leaf` as the minimum number.
    - If float, then `min_samples_leaf` is a fraction and
      `ceil(min_samples_leaf * n_samples)` are the minimum
      number of samples for each node.

    .. versionchanged:: 0.18
       Added float values for fractions.

min_weight_fraction_leaf : float, optional (default=0.)
    The minimum weighted fraction of the sum total of weights (of all
    the input samples) required to be at a leaf node. Samples have
    equal weight when sample_weight is not provided.

min_impurity_split : float, (default=1e-7)
    Threshold for early stopping in tree growth. A node will split
    if its impurity is above the threshold, otherwise it is a leaf.

    .. deprecated:: 0.19
       ``min_impurity_split`` has been deprecated in favor of
       ``min_impurity_decrease`` in 0.19. The default value of
       ``min_impurity_split`` will change from 1e-7 to 0 in 0.23 and it
       will be removed in 0.25. Use ``min_impurity_decrease`` instead.

class_weight : dict, list of dicts, "balanced" or None, default=None
    Weights associated with classes in the form ``{class_label: weight}``.
    If not given, all classes are supposed to have weight one. For
    multi-output problems, a list of dicts can be provided in the same
    order as the columns of y.

    Note that for multioutput (including multilabel) weights should be
    defined for each class of every column in its own dict. For example,
    for four-class multilabel classification weights should be
    [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of
    [{1:1}, {2:5}, {3:1}, {4:1}].

    The "balanced" mode uses the values of y to automatically adjust
    weights inversely proportional to class frequencies in the input data
    as ``n_samples / (n_classes * np.bincount(y))``

    For multi-output, the weights of each column of y will be multiplied.

    Note that these weights will be multiplied with sample_weight (passed
    through the fit method) if sample_weight is specified.

presort : bool, optional (default=False)
    Whether to presort the data to speed up the finding of best splits in
    fitting. For the default settings of a decision tree on large
    datasets, setting this to true may slow down the training process.
    When using either a smaller dataset or a restricted depth, this may
    speed up the training.

Attributes
----------
classes_ : array of shape = [n_classes] or a list of such arrays
    The classes labels (single output problem),
    or a list of arrays of class labels (multi-output problem).

feature_importances_ : array of shape = [n_features]
    The feature importances. The higher, the more important the
    feature. The importance of a feature is computed as the (normalized)
    total reduction of the criterion brought by that feature.  It is also
    known as the Gini importance [4]_.

max_features : int, float, string or None, optional (default=None)
    The number of features to consider when looking for the best split:

        - If int, then consider `max_features` features at each split.
        - If float, then `max_features` is a fraction and
          `int(max_features * n_features)` features are considered at each
          split.
        - If "auto", then `max_features=sqrt(n_features)`.
        - If "sqrt", then `max_features=sqrt(n_features)`.
        - If "log2", then `max_features=log2(n_features)`.
        - If None, then `max_features=n_features`.

    Note: the search for a split does not stop until at least one
    valid partition of the node samples is found, even if it requires to
    effectively inspect more than ``max_features`` features.

random_state : int, RandomState instance or None, optional (default=None)
    If int, random_state is the seed used by the random number generator;
    If RandomState instance, random_state is the random number generator;
    If None, the random number generator is the RandomState instance used
    by `np.random`.

max_leaf_nodes : int or None, optional (default=None)
    Grow a tree with ``max_leaf_nodes`` in best-first fashion.
    Best nodes are defined as relative reduction in impurity.
    If None then unlimited number of leaf nodes.

min_impurity_decrease : float, optional (default=0.)
    A node will be split if this split induces a decrease of the impurity
    greater than or equal to this value.

    The weighted impurity decrease equation is the following::

        N_t / N * (impurity - N_t_R / N_t * right_impurity
                            - N_t_L / N_t * left_impurity)

    where ``N`` is the total number of samples, ``N_t`` is the number of
    samples at the current node, ``N_t_L`` is the number of samples in the
    left child, and ``N_t_R`` is the number of samples in the right child.

    ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum,
    if ``sample_weight`` is passed.

    .. versionadded:: 0.19

max_features_ : int,
    The inferred value of max_features.

n_classes_ : int or list
    The number of classes (for single output problems),
    or a list containing the number of classes for each
    output (for multi-output problems).

n_features_ : int
    The number of features when ``fit`` is performed.

n_outputs_ : int
    The number of outputs when ``fit`` is performed.

tree_ : Tree object
    The underlying Tree object. Please refer to
    ``help(sklearn.tree._tree.Tree)`` for attributes of Tree object and
    :ref:`sphx_glr_auto_examples_tree_plot_unveil_tree_structure.py`
    for basic usage of these attributes.

Notes
-----
The default values for the parameters controlling the size of the trees
(e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and
unpruned trees which can potentially be very large on some data sets. To
reduce memory consumption, the complexity and size of the trees should be
controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore,
the best found split may vary, even with the same training data and
``max_features=n_features``, if the improvement of the criterion is
identical for several splits enumerated during the search of the best
split. To obtain a deterministic behaviour during fitting,
``random_state`` has to be fixed.

See also
--------
DecisionTreeRegressor

References
----------

.. [1] https://en.wikipedia.org/wiki/Decision_tree_learning

.. [2] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification
       and Regression Trees", Wadsworth, Belmont, CA, 1984.

.. [3] T. Hastie, R. Tibshirani and J. Friedman. "Elements of Statistical
       Learning", Springer, 2009.

.. [4] L. Breiman, and A. Cutler, "Random Forests",
       https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm
```