In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
plt.rcParams["figure.figsize"] = (10, 6)
```

Exercise 1

In this exercise, we will try different regression models on points drawn from three different analytical functions. We will see that the overall quality of the models heavily depends on the shape of the functions.

That said, let's define our three analytical functions.

In [3]:
```
def f1(x):
    return x * np.sin(x) + 2*x

def f2(x):
    return 10 * np.sin(x) + x**2

def f3(x):
    return np.sign(x) * (300 + x**2) + 20 * np.sin(x)
```

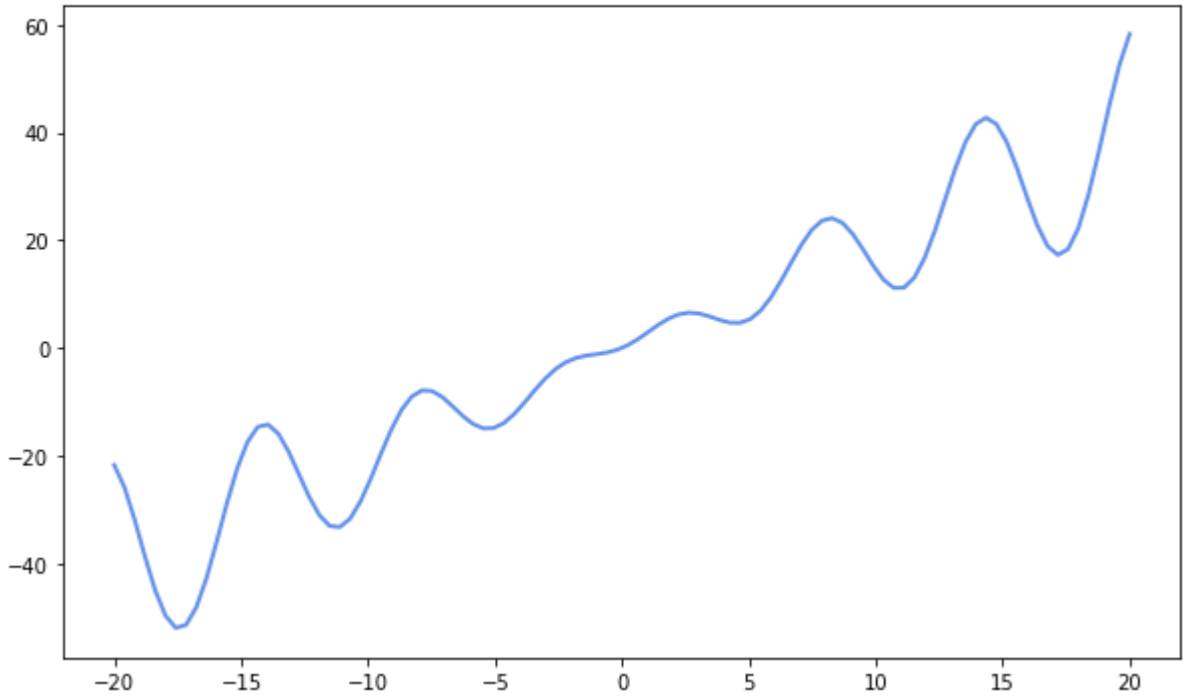
Exercises 1.1 and 1.2

As usual, we can define convenient functions to generate and plot our data.

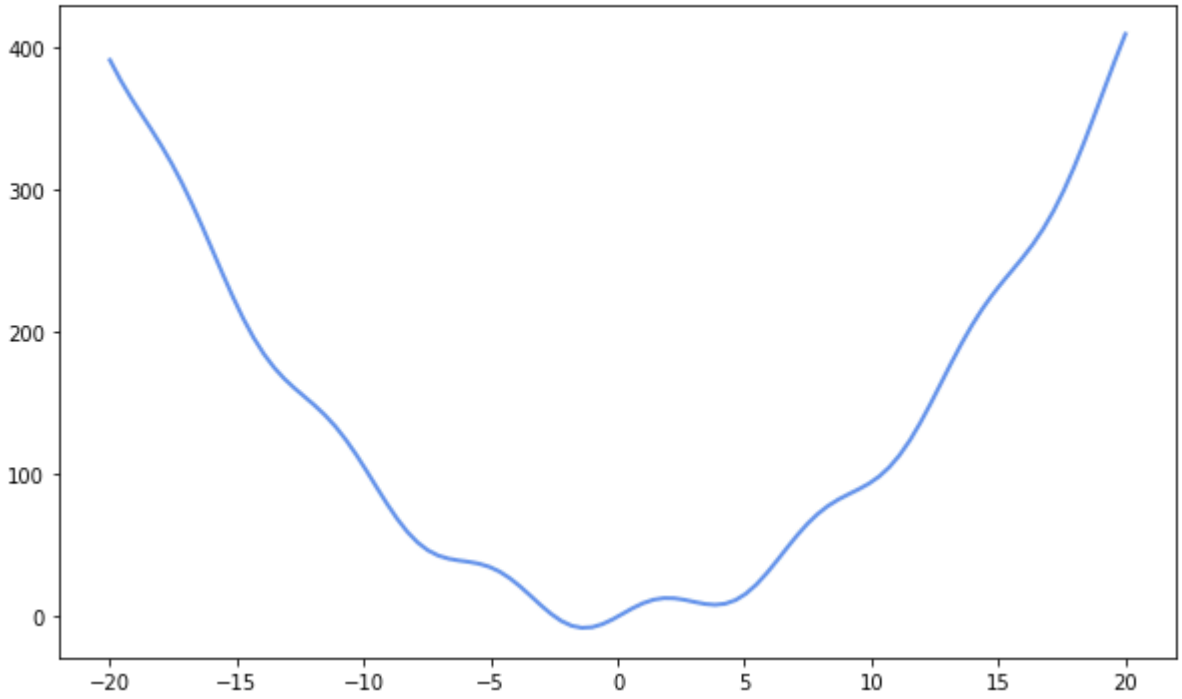
In [4]:
```
def generate_X_y(f):
    tr = 20
    n_samples = 100
    X = np.linspace(-tr, tr, n_samples)
    y = f(X)
    return X, y

def plot_f(X, y, title):
    LW = 4
    fig, ax = plt.subplots()
    ax.plot(X, y, color='cornflowerblue', linewidth=.5*LW, label="ground truth")
    fig.suptitle(title)
    #ax.scatter(X_train, y_train, color='navy', s=30, marker='o', label="training points")
```

In [5]:
```
for f in [f1, f2, f3]:
    X, y = generate_X_y(f)
    plot_f(X, y, f)
```

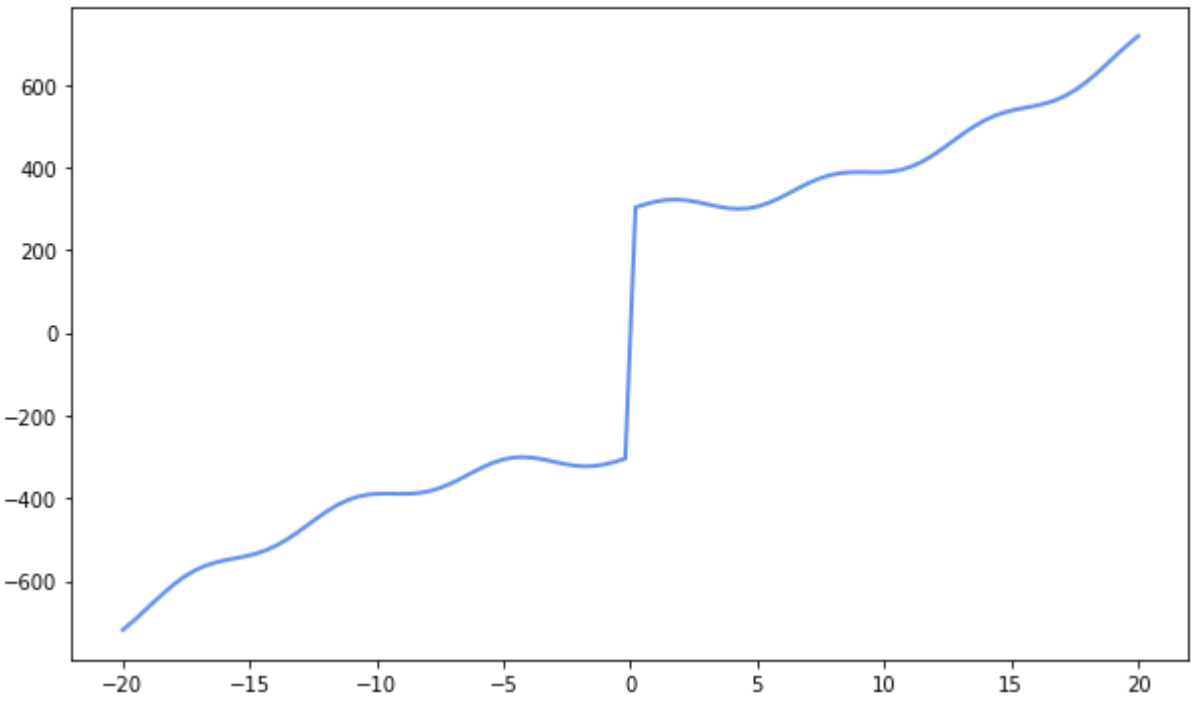
<function f1 at 0x7ffe54e2a158>



<function f2 at 0x7ffe54e2a0d0>



<function f3 at 0x7ffe54e2a2f0>



f1 has two main components: a sine wave with an increasing magnitude (due to the factor *x*), plus an additive factor that gives a non zero slope to the curve.

f2 has a sine wave with fixed amplitude, modulated by a parabolic function.

f3 presents a sine wave modulated by a parabolic function. However, it is different from f2. The quadratic expression changes concativity (i.e. there is an inflection point) at zero, due to the *sign* component. Also, the factor *sign(x) 300* produces a discontinuity point of type one.

The shape of the functions tells us that f2 and f3 have an infinite asymptotic value, for both positive and negative values. Hence, they can be approximated with polynomial regressors. However, f3 has a discontinuity point at 0, which can harden the approximation for classifiers. Even if the ordinary least squares regressor would fit the linear trend present in f1, we can see that all the three functions cannot be approximated with a linear regressor, with sufficient results, for large values of *x*.

Back to programming, you can notice how the functions behave as simple objects in Python. It is not unsual to assign them to variables or, like in this case, create an array of them. Here, the *f* variable points, at each loop cycle, to a different function in memory, and gets invoked as a callable object. Additionally, you can note the printed version of *f* contains the name of the assigned function and the memory address where its code is stored.

Exercises 1.3, 1.4, and 1.5

Let's now fit our regression models. To do so, we define a function to create the training and test points given a function and a scikit-learn Pipeline to apply to them. For the seek of readability, we inspect one analytic function at a time.

In [6]:
```
from sklearn.model_selection import train_test_split

def generate_train_test(f, X, y):
    X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                        train_size=30,
                                                        random_state=42,
                                                        shuffle=True)
    y_test = y_test[X_test.argsort()]
    X_test.sort()
    return X_train, X_test, y_train, y_test
```

Note that here the cardinality of training and test sets are reversed with respect to common cases. Typically, the 70% of the dataset is kept as training set and the remaining 30% is used to test the model.

In the following cells, we test different regression algorithm and evaluate the regression error through two metrics: the MSE and the R2 score. Thus, we use a convenient Python function that, given an analytical function, a dataset, and a model, produces the value of the two metrics. To better understand the quality of each regressor, the function provides also a graphical representation of the predicted values against the real values drawn from the original curve.

In [7]:
```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score

def evaluate_model(f, X, y, model, model_name):
    X_train, X_test, y_train, y_test = generate_train_test(f, X, y)

    # plot the real function and the training points
    LW = 2
    fig, ax = plt.subplots()
    ax.plot(X, y, color='cornflowerblue', linewidth=.5*LW, label="ground tru
th")
    ax.scatter(X_train, y_train, color='navy', s=30, marker='o', label="trai
ning points")

    # predict the test points and plot them onto the chart
    model.fit(X_train.reshape(-1, 1), y_train)
    y_hat = model.predict(X_test.reshape(-1, 1))
    ax.plot(X_test, y_hat, linewidth=LW, label=name, color='r')

    fig.suptitle(f"{f} approximated by {model_name}")
    fig.legend()

    return mean_squared_error(y_test, y_hat), r2_score(y_test, y_hat)
```

In [8]:
```
from sklearn.linear_model import LinearRegression
from sklearn.neural_network import MLPRegressor
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import Ridge
from sklearn.preprocessing import FunctionTransformer, PolynomialFeatures
from sklearn.compose import make_column_transformer
from sklearn.pipeline import make_pipeline
```

The performance of each regressor are collected and displayed with the library PrettyTable (https://github.com/jazzband/prettytable).

In [9]:

```
from prettytable import PrettyTable

degree = 5
models = [
    LinearRegression(),
    Ridge(random_state=42),
    MLPRegressor(hidden_layer_sizes=(10,), random_state=42, max_iter=10000),
    MLPRegressor(hidden_layer_sizes=(10,10),  activation='tanh', solver='lbf
gs',
                 alpha=0.000, batch_size='auto', learning_rate='constant',
                 learning_rate_init=0.01, power_t=0.5, max_iter=10000, shuff
le=True,
                 random_state=42, tol=0.0001, verbose=True, warm_start=Fals
e,
                 momentum=0.0, nesterovs_momentum=False, early_stopping=Fals
e,
                 validation_fraction=0.0, beta_1=0.9, beta_2=0.999, epsilon=
1e-08),
    SVR(gamma='scale'),
    RandomForestRegressor(n_estimators=300),
    make_pipeline(
        make_column_transformer(
            (FunctionTransformer(np.sin), [0]),
            (PolynomialFeatures(degree), [0])
        ),
        LinearRegression()
    ),
    make_pipeline(
        make_column_transformer(
            (FunctionTransformer(np.sin), [0]),
            (PolynomialFeatures(degree), [0])
        ),
        Ridge(alpha=1)
    )
]

names = [
    'linreg',
    'ridge',
    'mlp_standard',
    'mlp_tuned',
    'svr',
    'rf',
    f'sin+poly{degree}+linreg',
    f'sin+poly{degree}+ridge'
]
```

Before the actual simulation, let's spend a few comments on the code above. We generated a list with a few models to be tested along with a respective textual representation that is used in the title of the each figure. The single-stage models are:

- a simple Linear Regression model;
- a MultiLayer Perceptron (a.k.a. feed forward neural network) with a single hidden layer with a reasonable number of hidden nodes;
- a deeper MultiLayer Perceptron with tuned parameters. We do not spend too much time on them since it goes beyond the scope of the exercise. However, the provided paramenters should let you grasp how wide are the tuning possibilities of this estimator;
- a Support Vector Regressor (gamma = 'scale' is recommended for the scikit-learn implementation);
- a Random Forest Regressor (the higher the number of estimator, the better).

We also adopted two composite pipelines. Each of them has two steps:

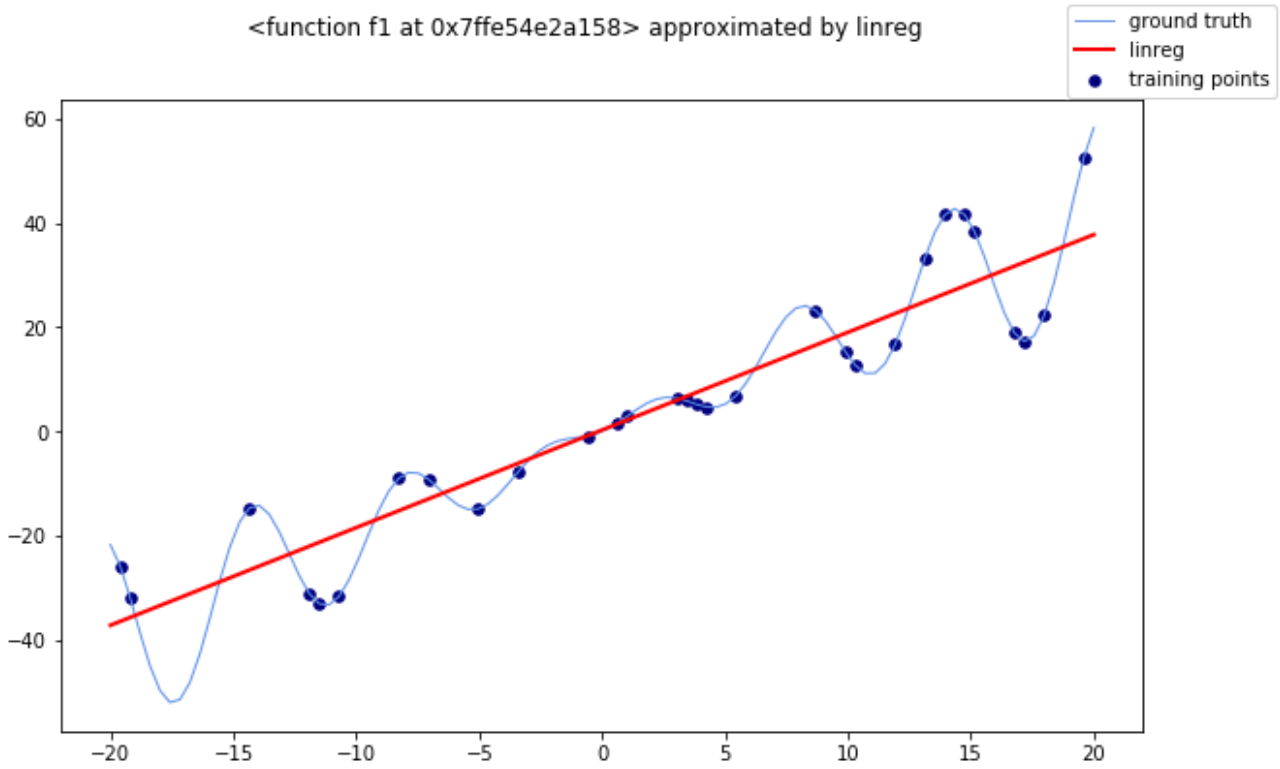
1. in the first step we build a ColumnTransformer (https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html#sklearn.compose.ColumnTransforme using an utility method. The Transform objects in scikit-learn's preprocessing module are typically used to perfom some sort of transformation of the input columns (e.g. scaling and normalization, application of a function). The ColumnTransform lets you specify a series of single Transformers to apply to columns of your choice. In our case, we apply two types of column transformation to the first (and only) column (see the parameter [0] of each tuple). One uses a FunctionTransformer (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.FunctionTransformer.html#sklearn.preprocessing.Function to generate one additional feature in the form *sin(x)*. The other creates new polynomial features using the pattern provided by the class PolynomialFeatures (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html#sklearn.preprocessing.Polynomi
2. In the second test we apply the regressor model as usual. Specifically, we test LinearRegression a Ridge again to measure the impact of the previous preprocessing on the performance.

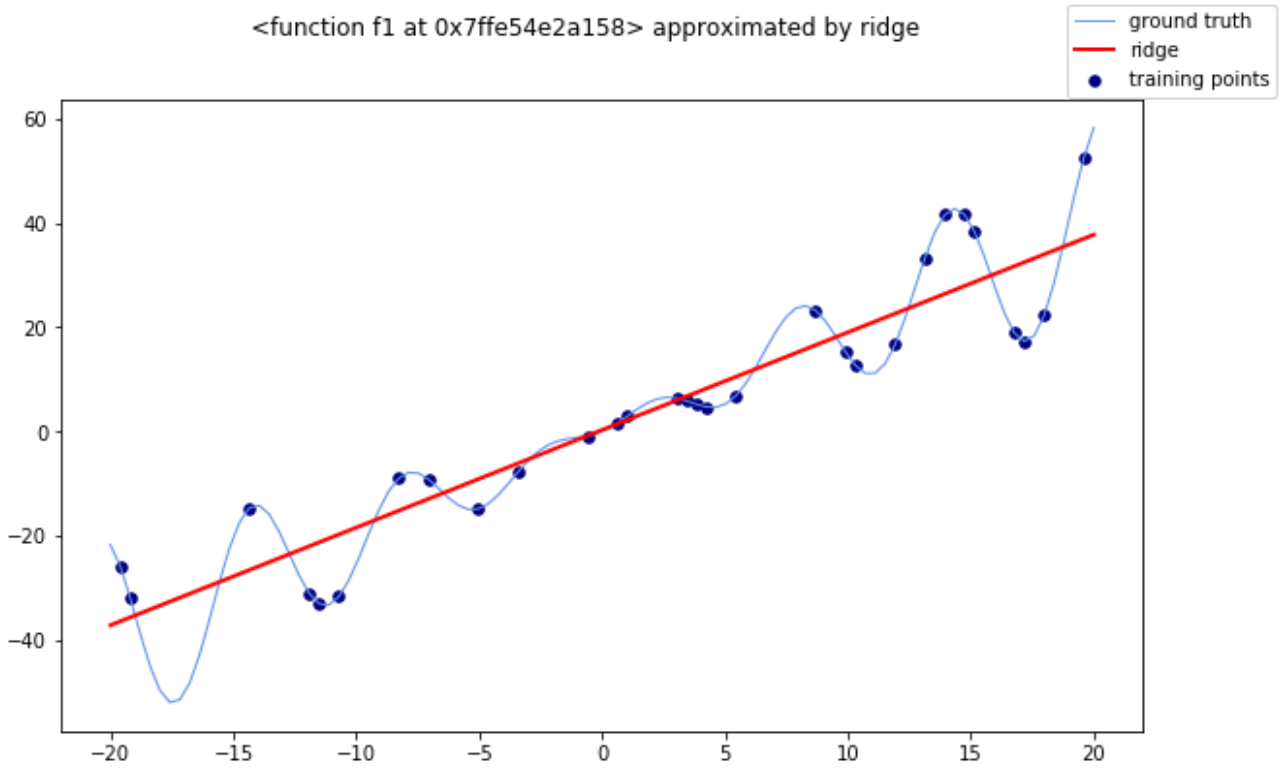
Results for f = f1

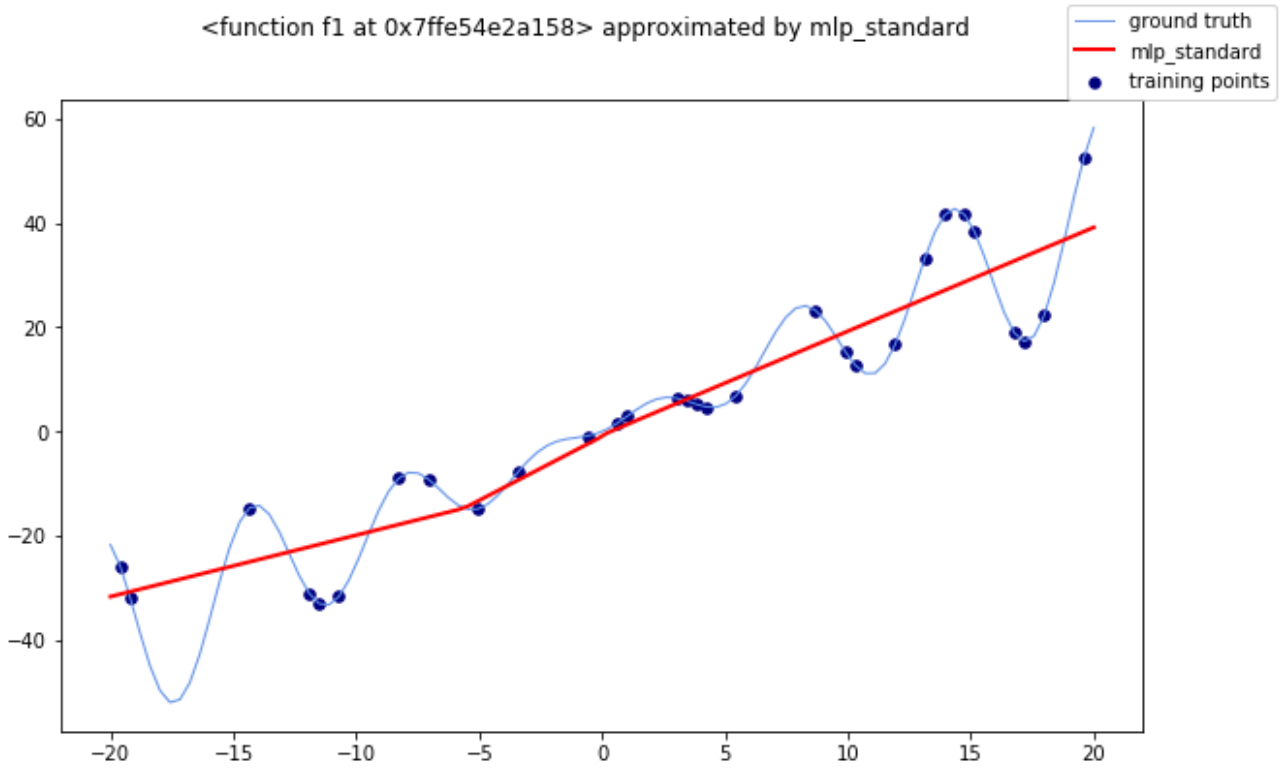
In [10]:

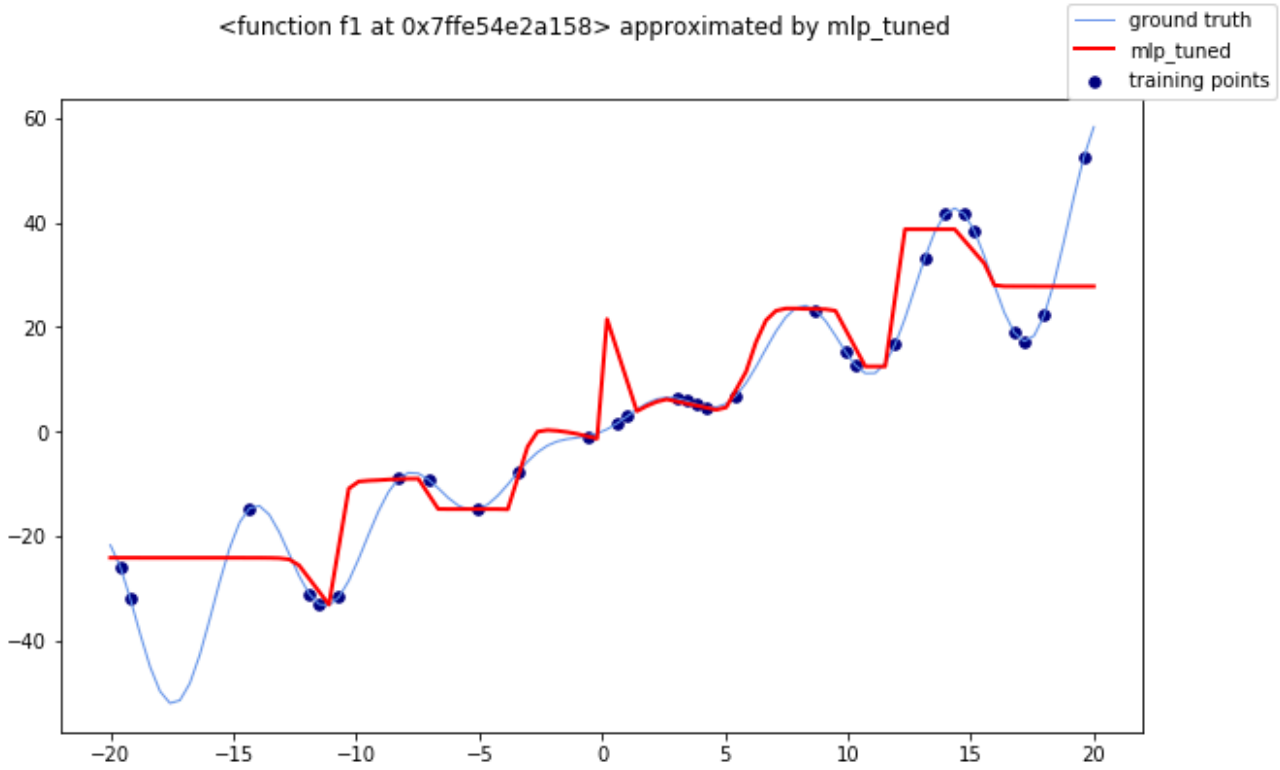
```
t = PrettyTable()
t.field_names = ['model', 'MSE', 'R2']
X, y = generate_X_y(f1)
for model, name in zip(models, names):
    mse, r2 = evaluate_model(f1, X, y, model, name)
    t.add_row([name, mse, r2])

print(t)
```

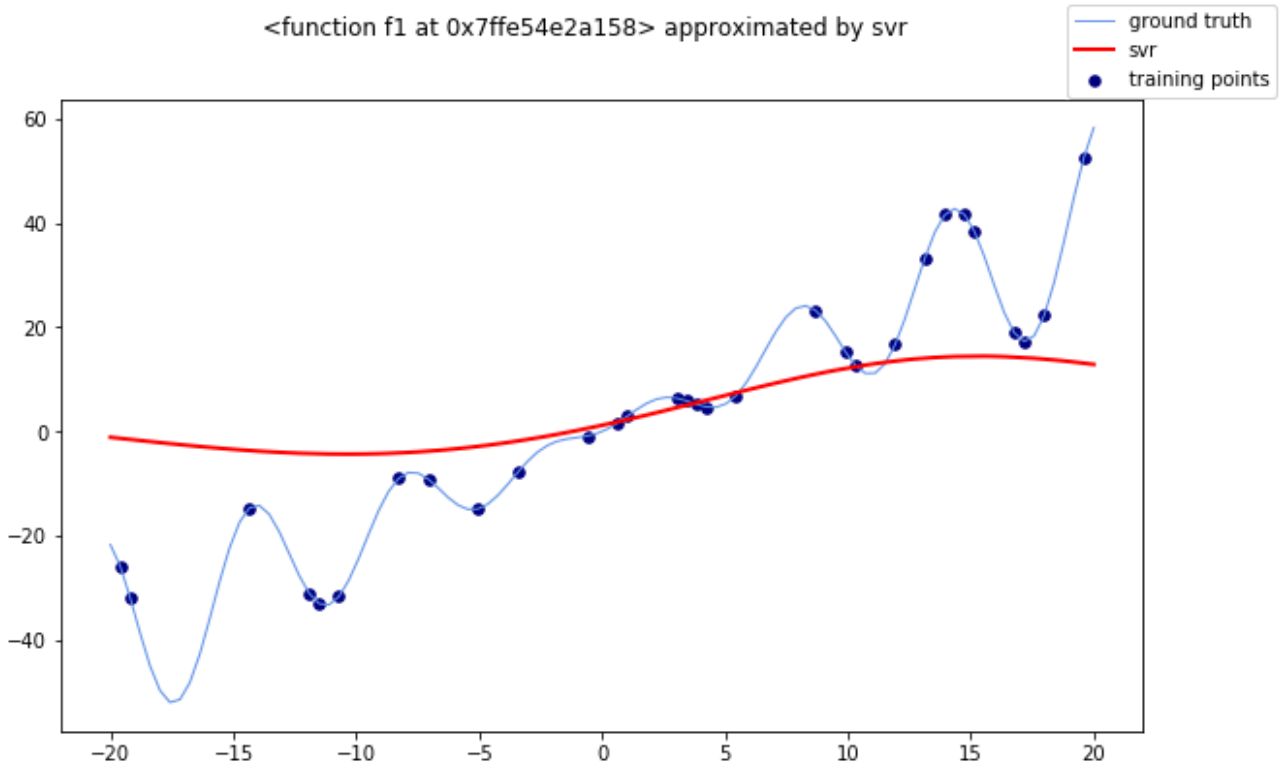
model	MSE	R2
linreg	65.72227592689515	0.8950661957810067
ridge	65.74765554283465	0.8950256740612001
mlp_standard	75.46463406818695	0.879511306857702
mlp_tuned	104.46617412806707	0.8332067338073432
svr	399.9552743551347	0.36142155968215894
rf	57.024236887693924	0.9089536991085534
sin+poly5+linreg	95.50251981840506	0.8475183250167385
sin+poly5+ridge	93.41072390557375	0.8508581378836467

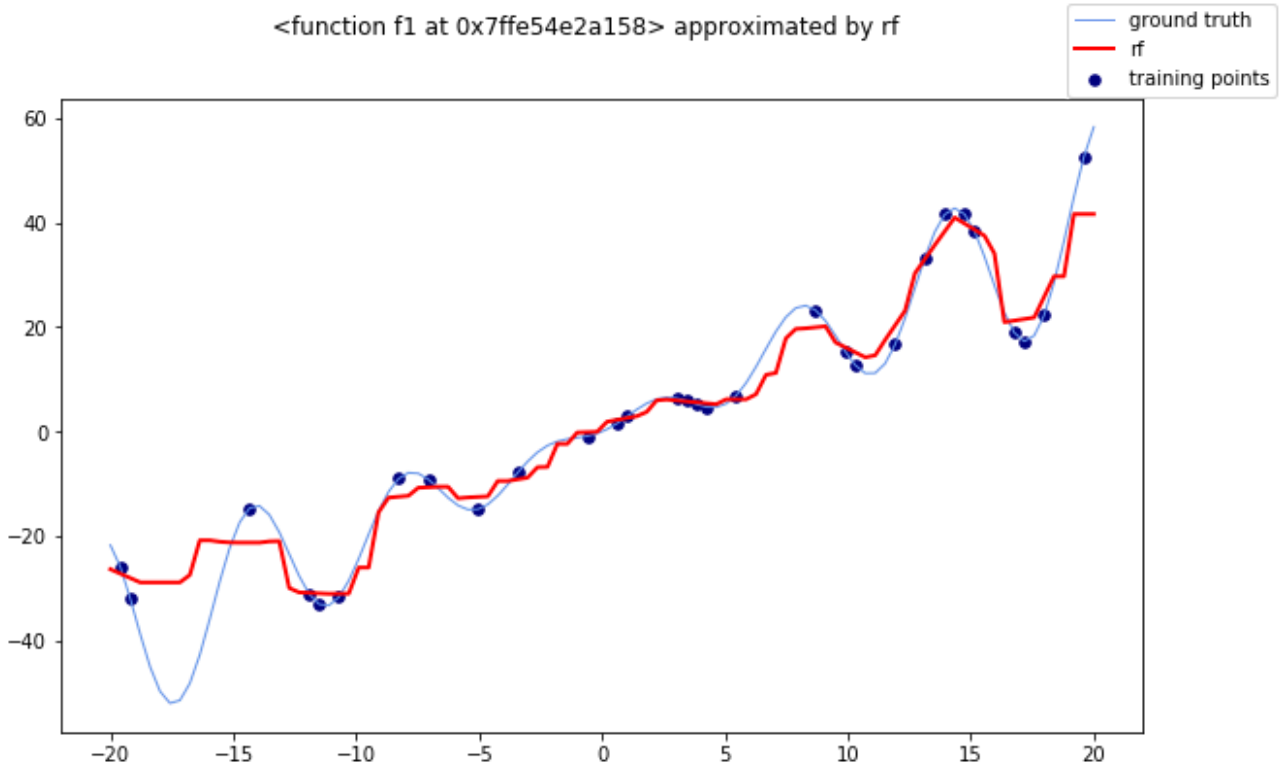


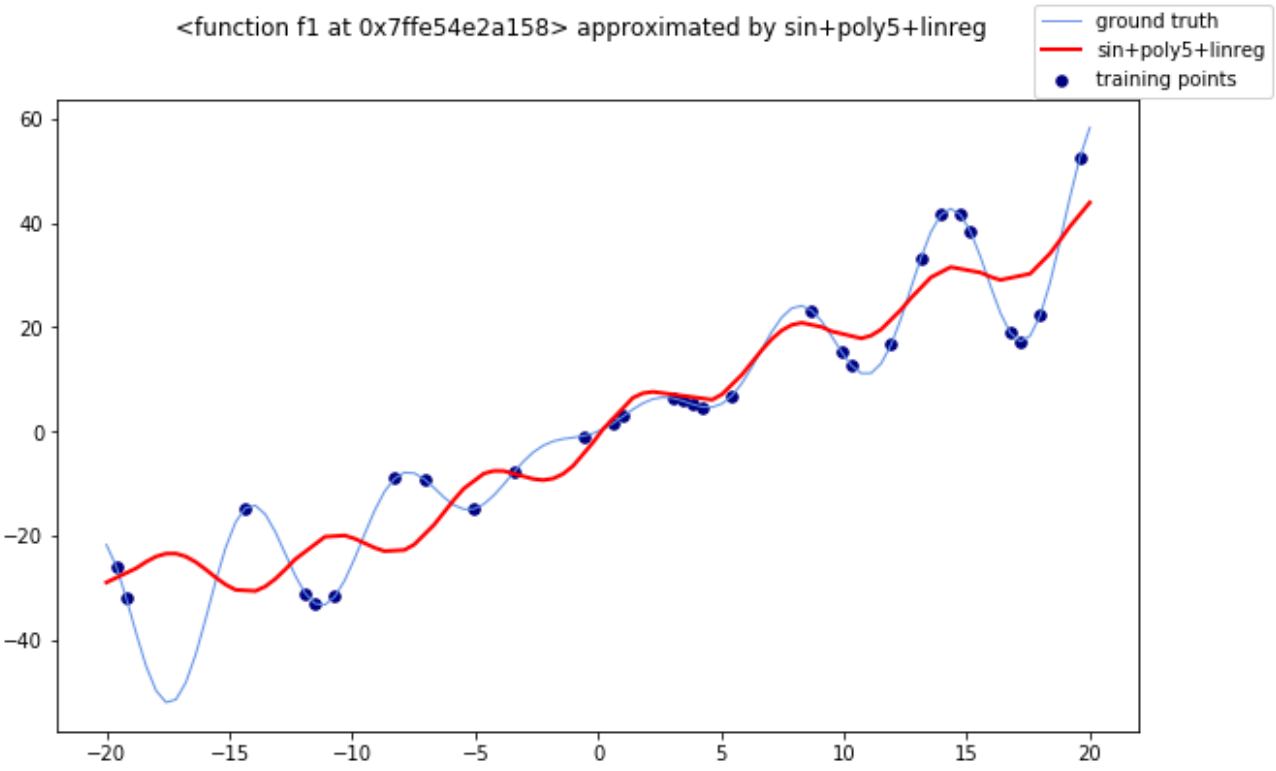


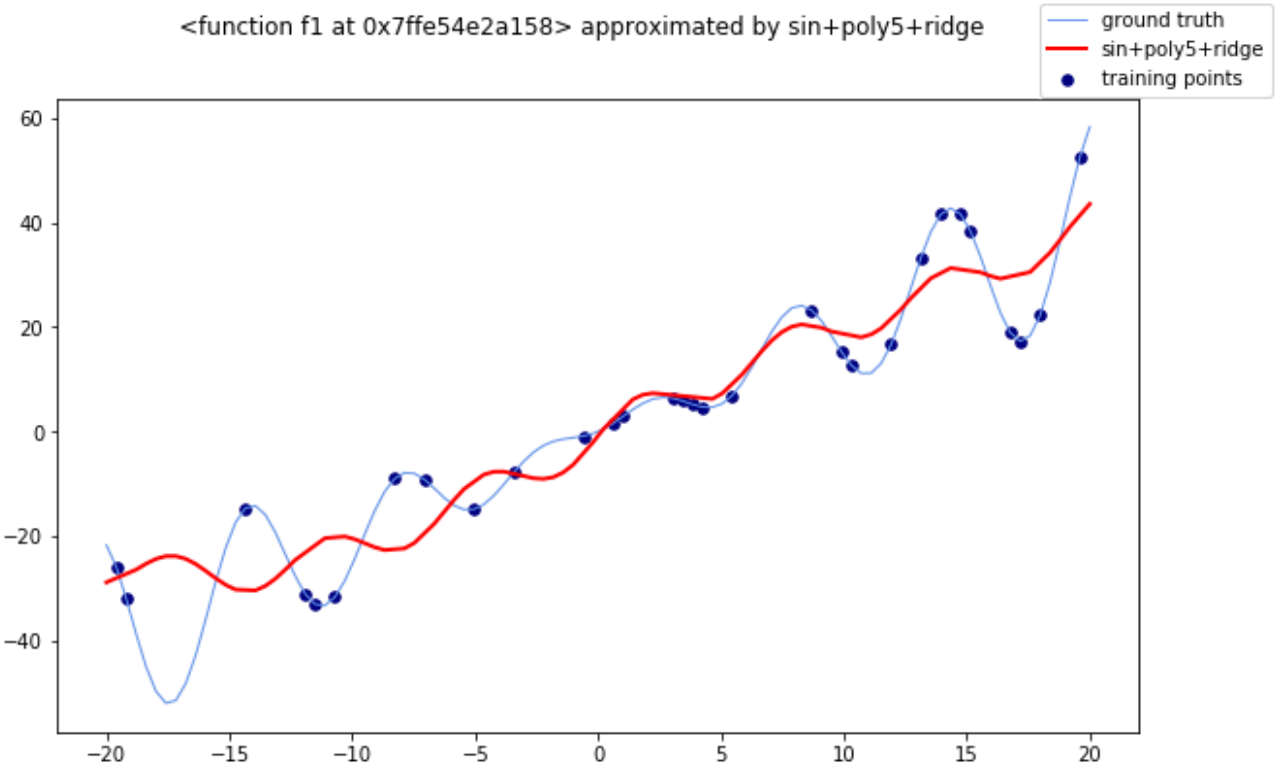












We see that LinearRegression and Ridge have comparable results. SVR fails at modeling the correct shape of the curve, the standard MLP converged to an approximation of the linear behavior, while the tuned MLP shows a more complex pattern, although an abnormal spike near zero worsen probably affects negatively the performance. The Random Forest regressor achieved the best performance in both the MSE and R2 scores. Finally, it is worth noting that the addition of *sin(x)* and the polynomial features up to the fifth degree slightly worsened the performance of LinearRegression and Ridge. Having the real function shape, we can graphically find a motivation for that. For both the regressors, the predictions (the red curve) are able follow a sinusoidal pattern (which we might have expected by the introduction of *sin(x)*). Nonetheless, for negative values of x, this curve is in couterphase with the real one, increasing the overall error.

Exercise 1.6

In many real tasks, we typically suppose that the measurements of the predictive variables carry some sort of noise. To reflect this aspect to our synthetic data, we can inject it manually.

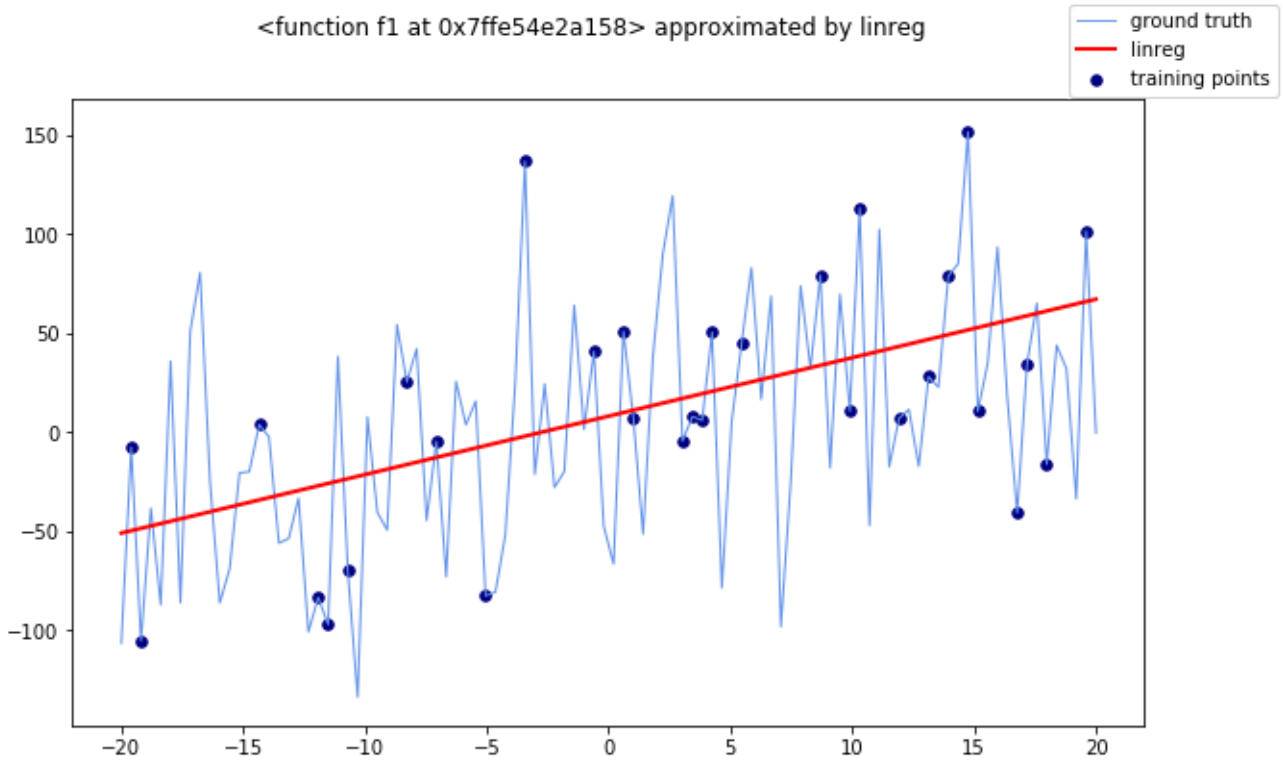
In [11]:
```
def inject_noise(x):
    """Add a random noise drawn from a normal distribution."""
    return x + np.random.normal(0, 50, size=x.size)
```

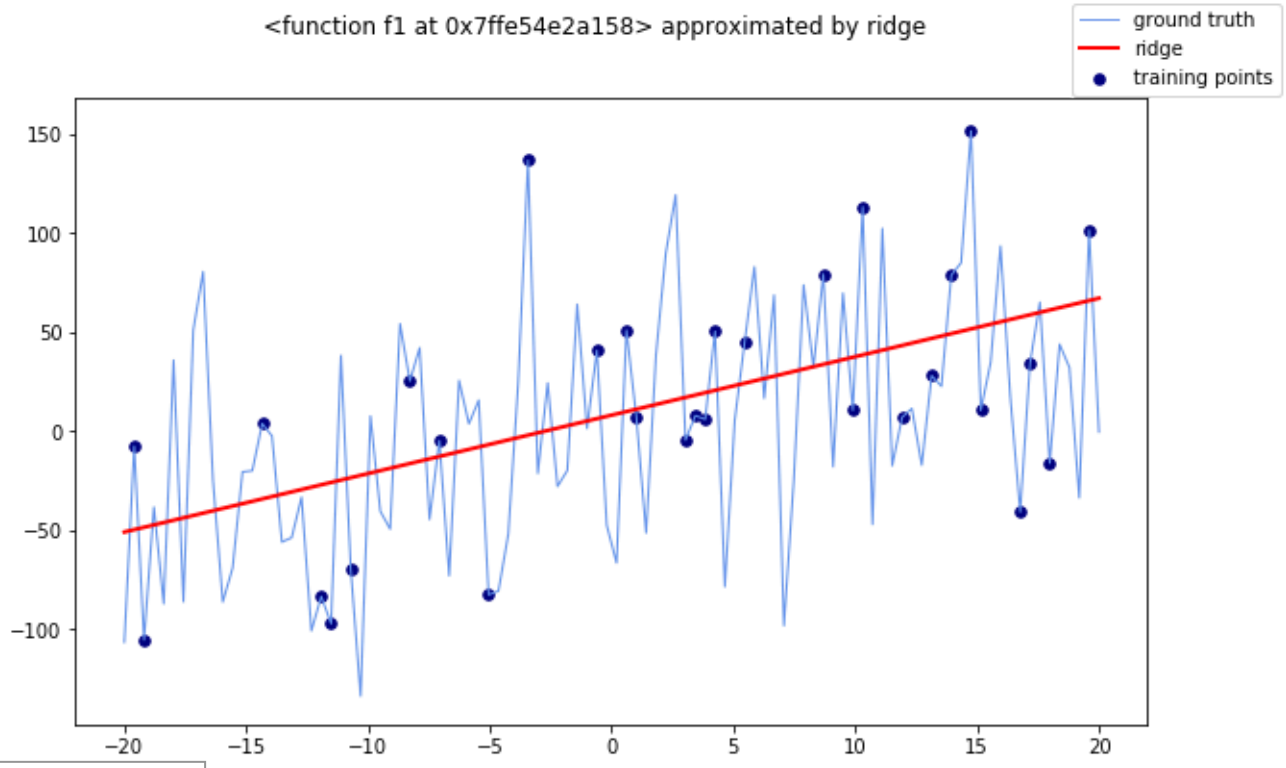
In [12]:

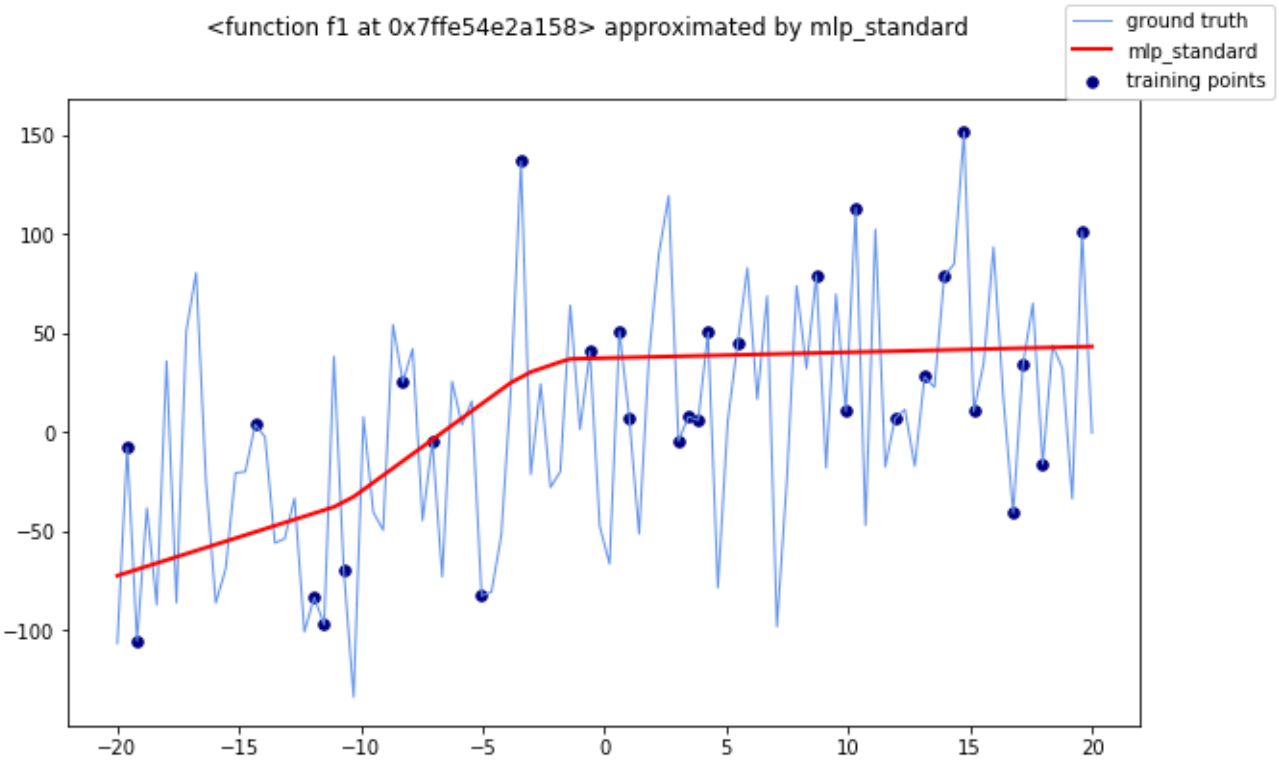
```
X, y = generate_X_y(f1)
y = inject_noise(y)

t = PrettyTable()
t.field_names = ['model', 'MSE', 'R2']
for model, name in zip(models, names):
    mse, r2 = evaluate_model(f1, X, y, model, name)
    t.add_row([name, mse, r2])

print(t)
```

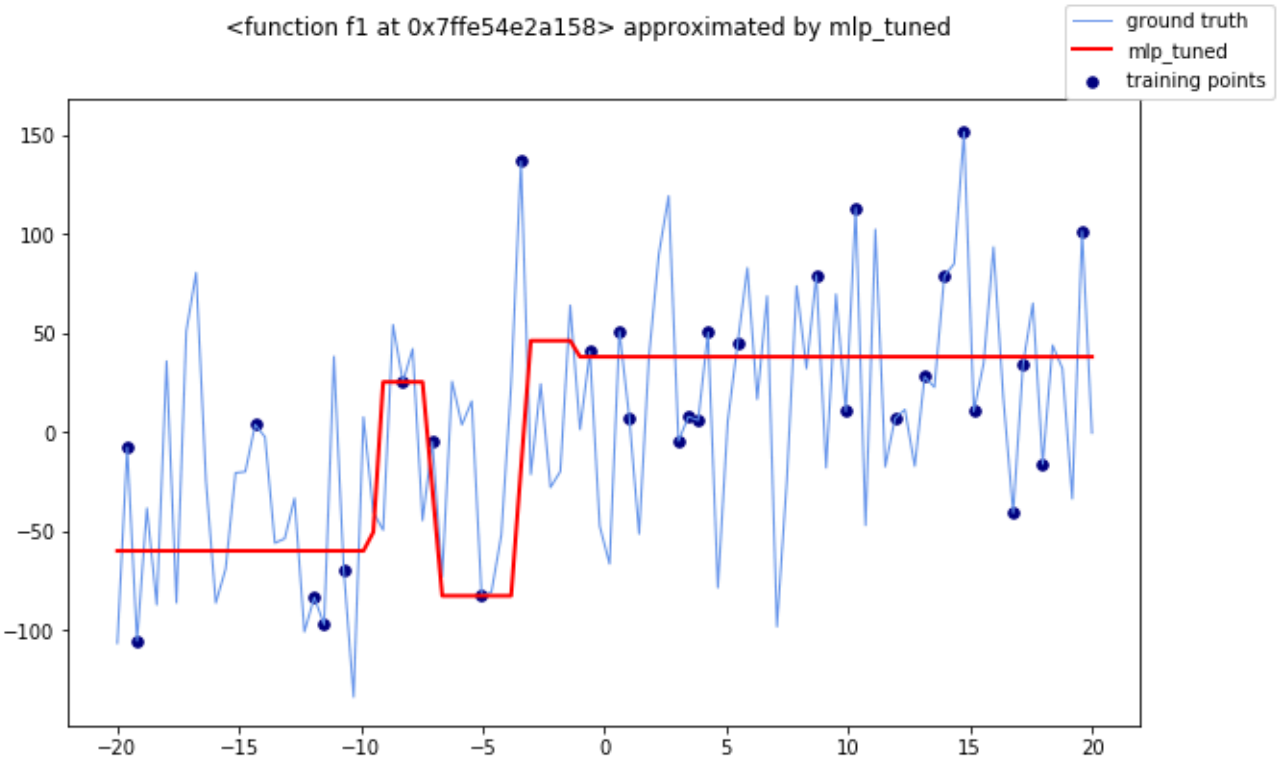
```
/Users/giuseppe/miniconda3/lib/python3.6/site-packages/sklearn/neural_network/_multilayer_perceptron.py:571: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (10000) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
```

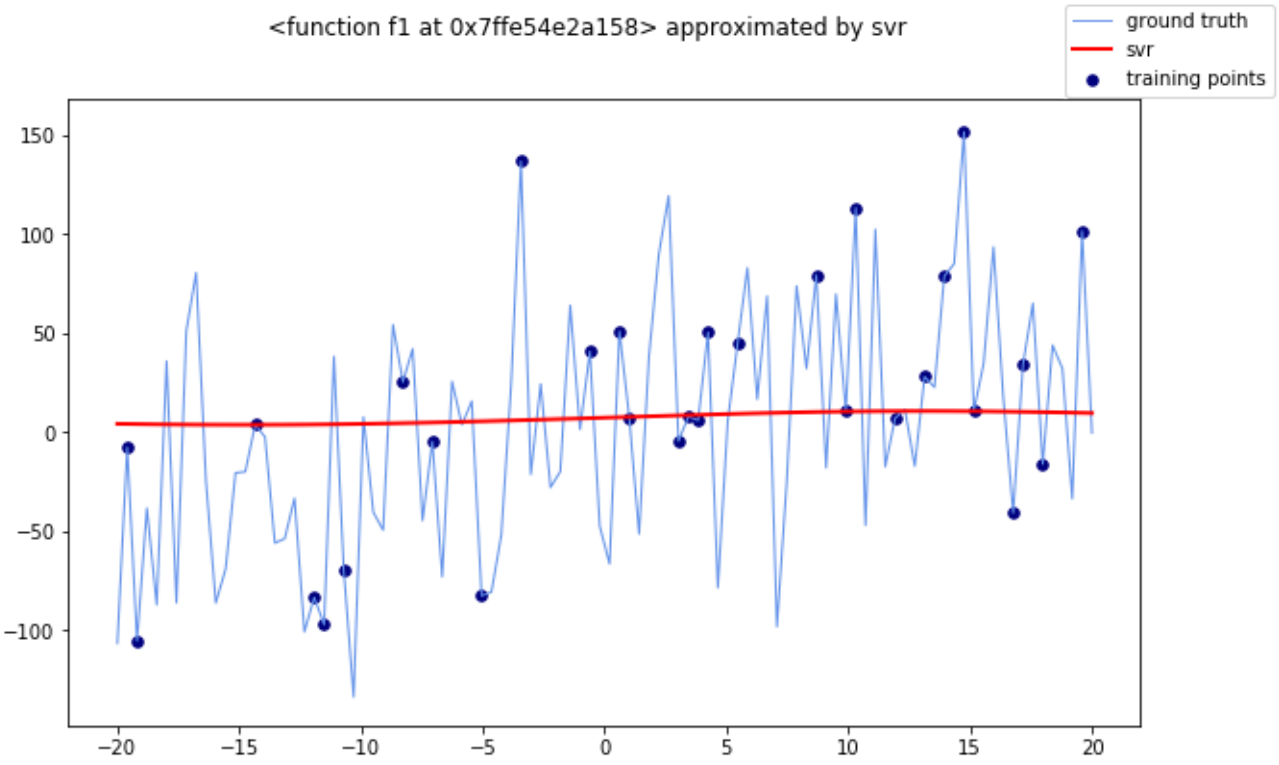
model	MSE	R2
linreg	2957.236994778774	0.11280320127342303
ridge	2957.0994988641382	0.11284445124274178
mlp_standard	3296.3441666075573	0.011068102665194646
mlp_tuned	3688.7875228403523	-0.10666831479001027
svr	3324.973510100103	0.002479050810008321
rf	4556.727445167173	-0.36705783444464113
sin+poly5+linreg	3288.756662982372	0.013344419693040566
sin+poly5+ridge	3289.5728746771806	0.01309954909119293

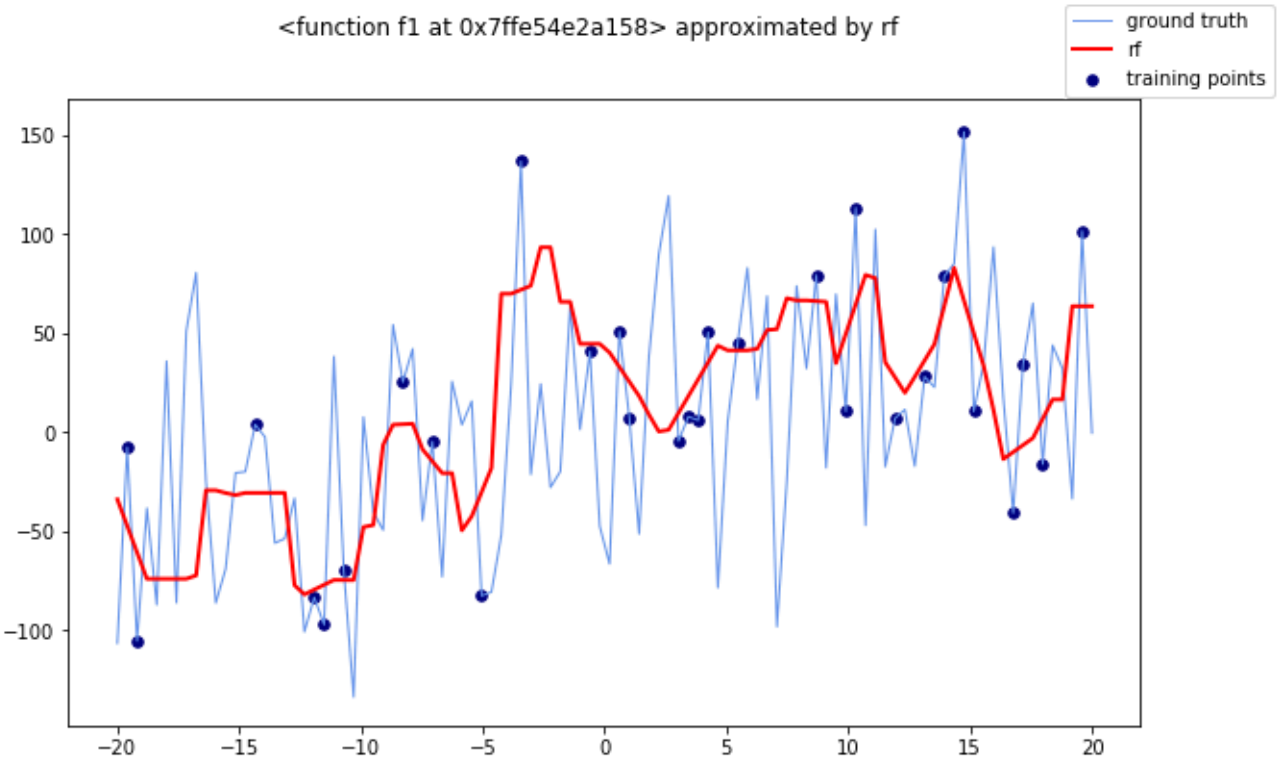


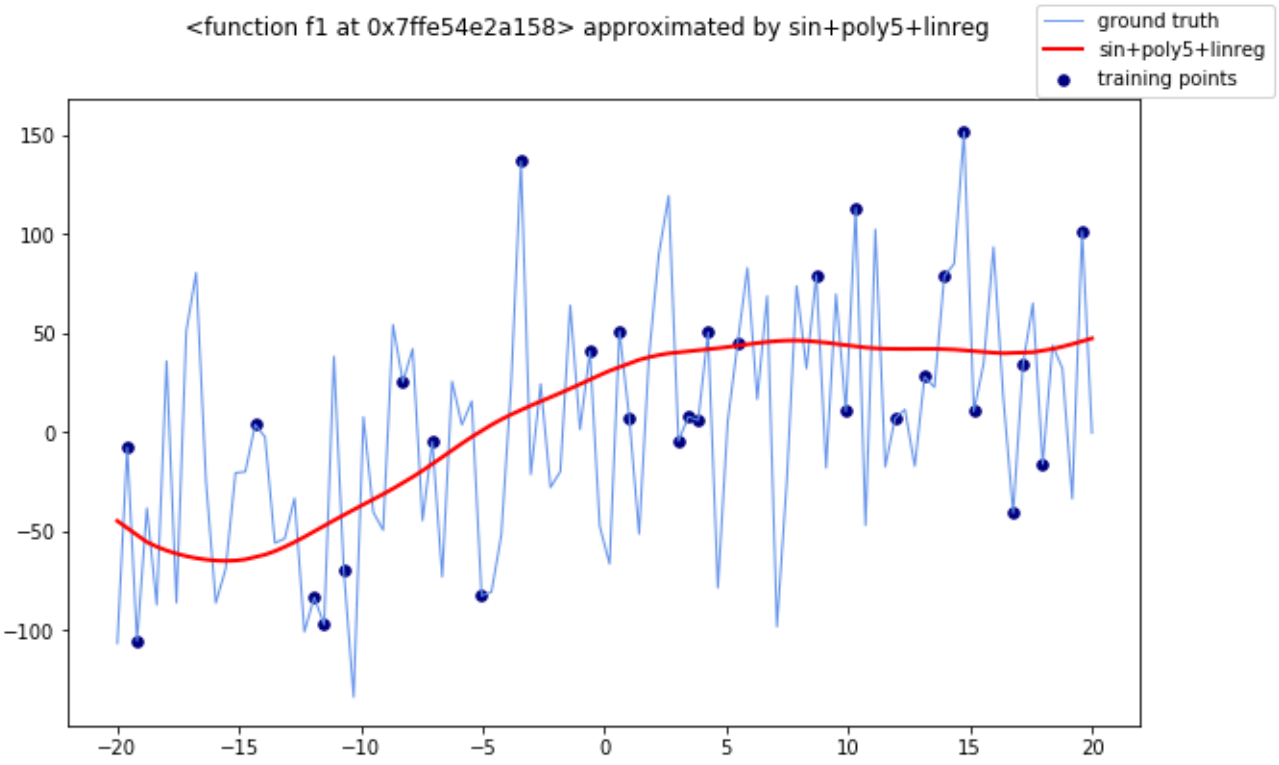


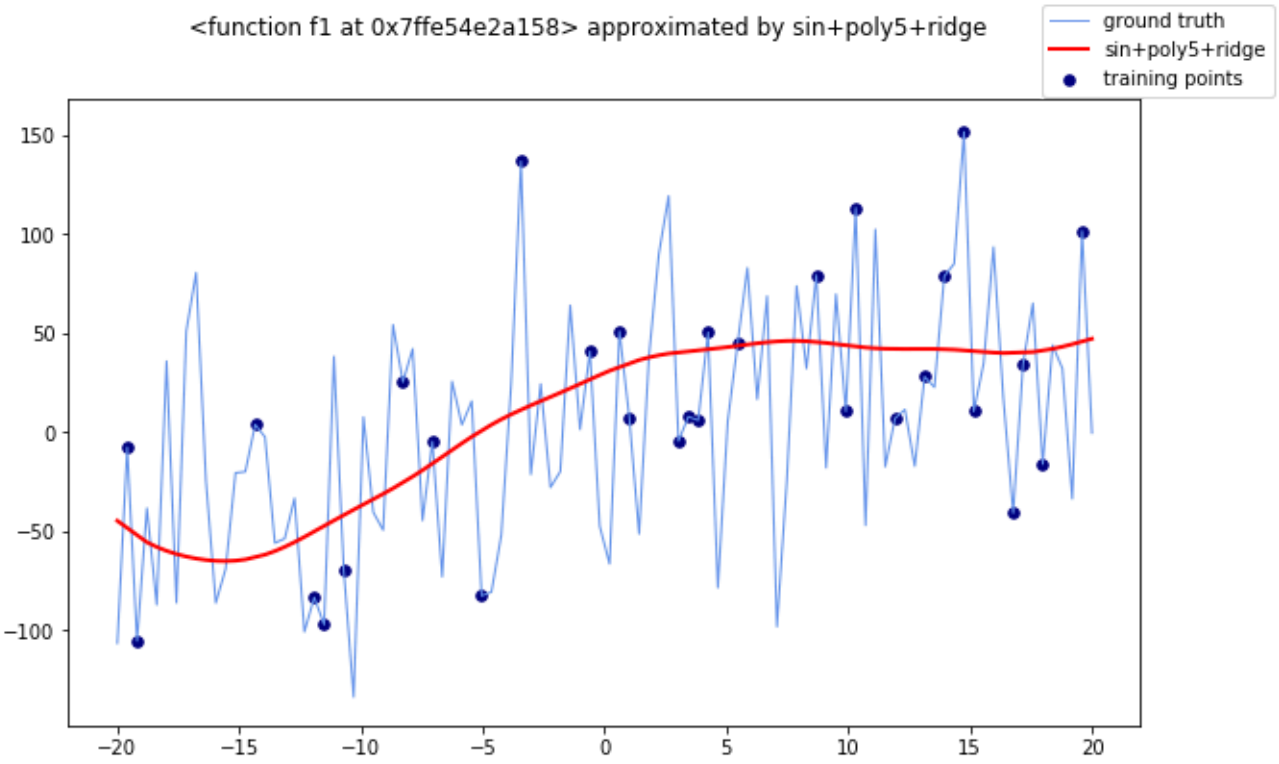












This kind of noise (normal, standard deviation=50) makes the problem extremely more complex. The initial shape of the function is lost and the performace are worse for any classifier.

Note: the reported warning concerns MLP. Even with 10000 iterations is is not able to converge to stable values for the weights in the network.

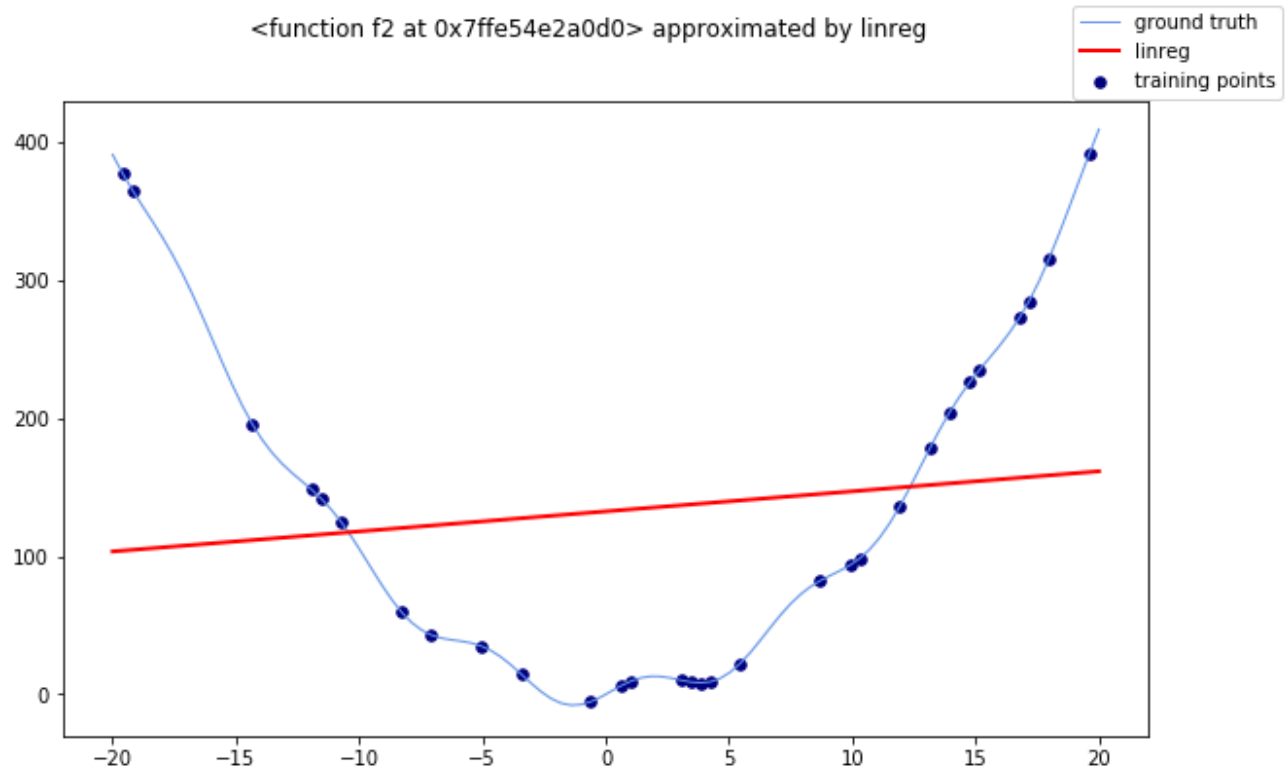
Results for f = f2

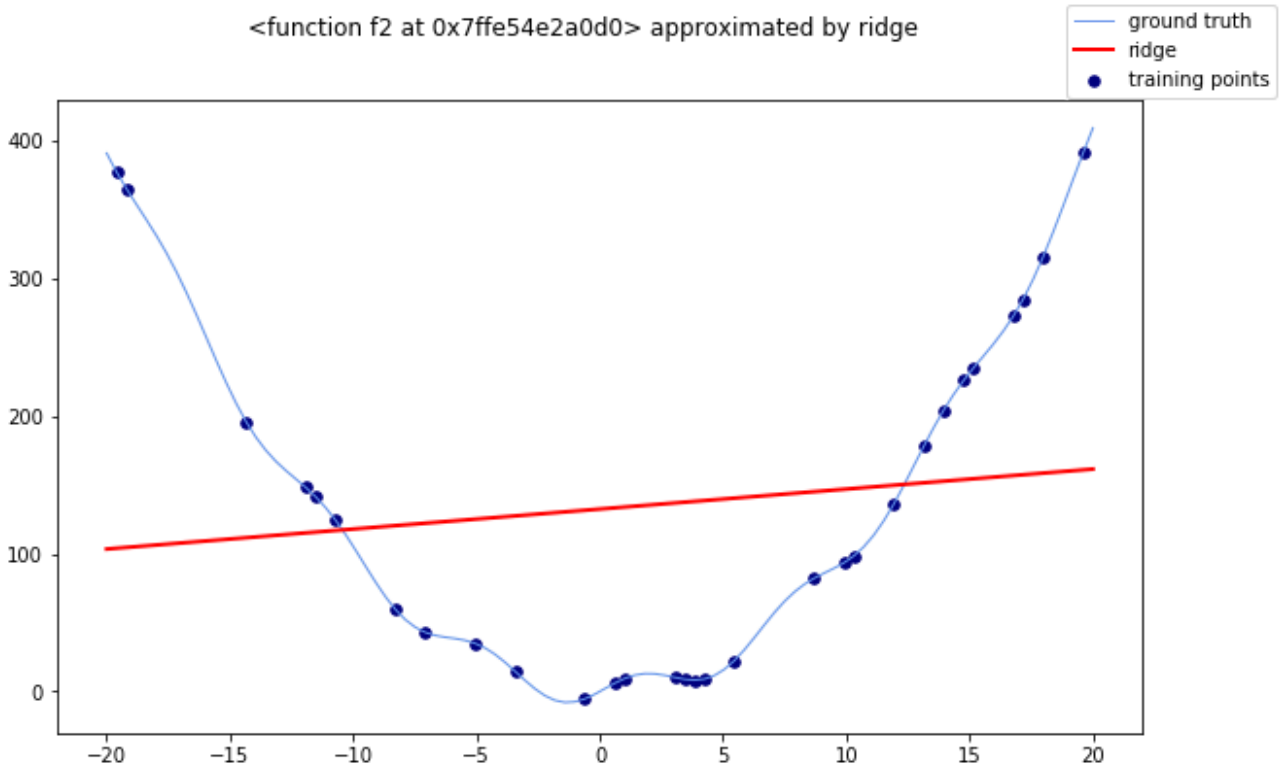
In [13]:

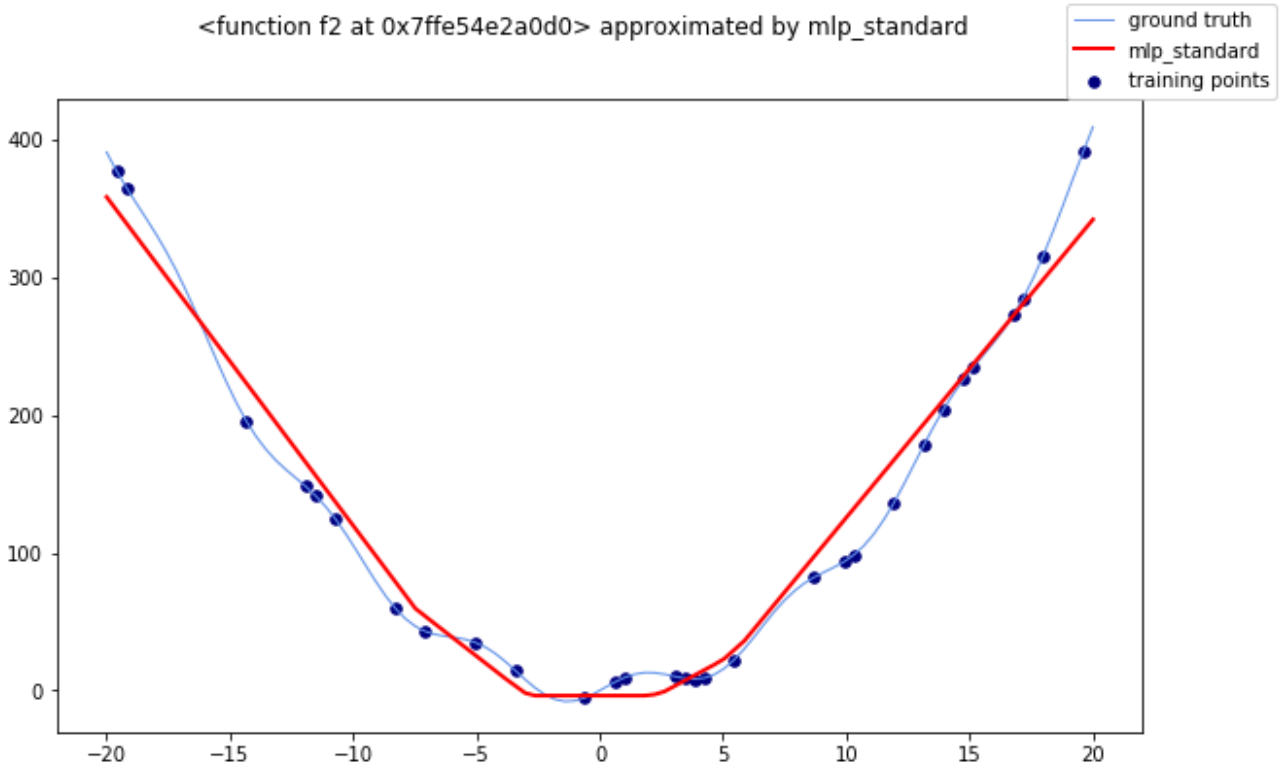
```
t = PrettyTable()
t.field_names = ['model', 'MSE', 'R2']
X, y = generate_X_y(f2)
for model, name in zip(models, names):
    mse, r2 = evaluate_model(f2, X, y, model, name)
    t.add_row([name, mse, r2])

print(t)
```

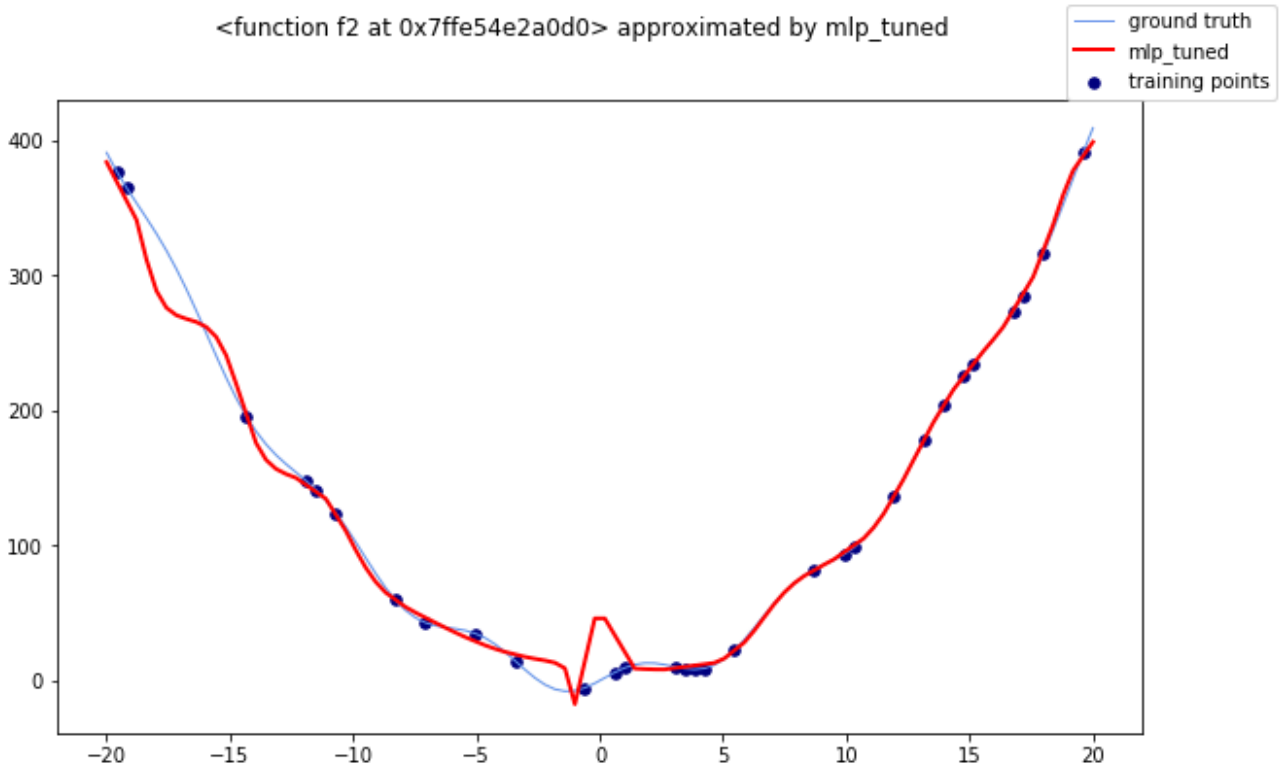
```
/Users/giuseppe/miniconda3/lib/python3.6/site-packages/sklearn/neural_networ
k/_multilayer_perceptron.py:571: ConvergenceWarning: Stochastic Optimizer: M
aximum iterations (10000) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
/Users/giuseppe/miniconda3/lib/python3.6/site-packages/sklearn/neural_networ
k/_multilayer_perceptron.py:470: ConvergenceWarning: lbfgs failed to converg
e (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  self.n_iter_ = _check_optimize_result("lbfgs", opt_res, self.max_iter)
```

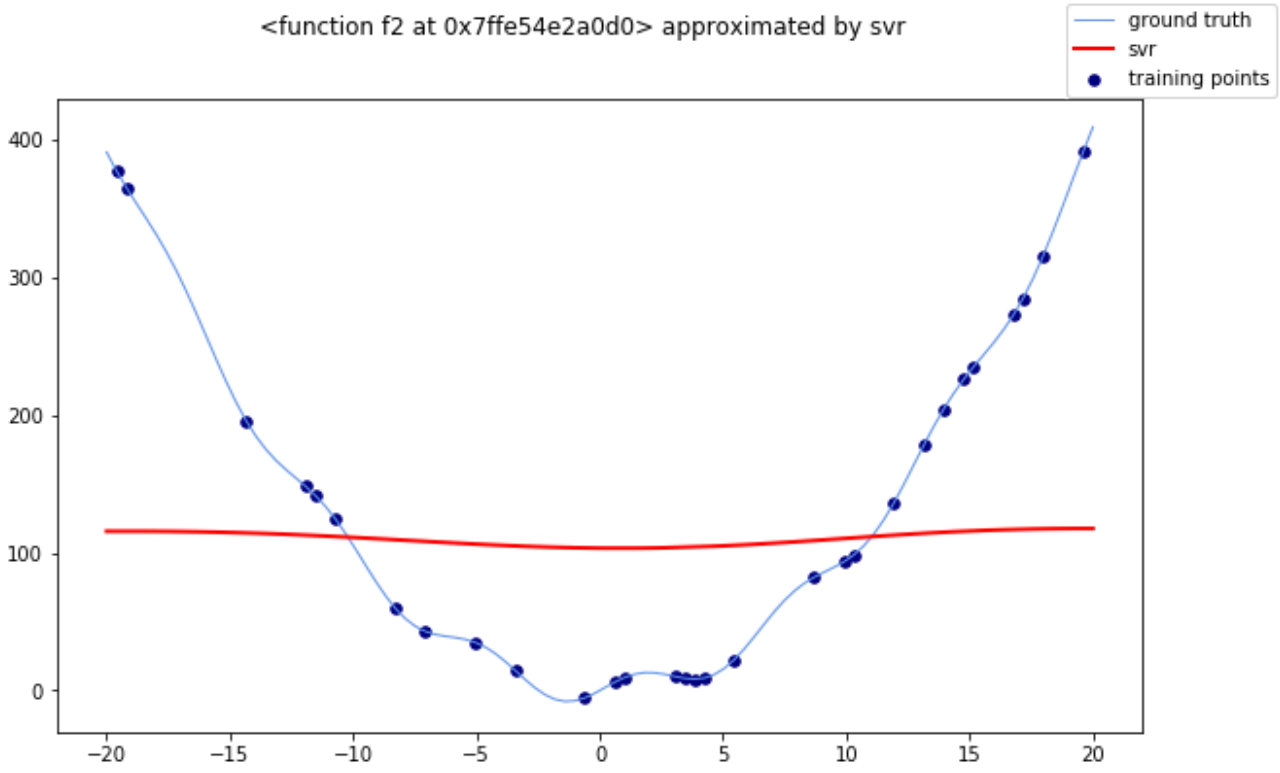
model	MSE	R2
linreg	15300.007189428956	-0.037420556511206016
ridge	15299.785419953309	-0.037405519380177665
mlp_standard	383.373251167696	0.9740052872751136
mlp_tuned	194.42309666497553	0.9868170965775644
svr	14341.447317768201	0.027574819189968847
rf	229.9094464786911	0.9844109363505463
sin+poly5+linreg	5.419623147322589e-23	1.0
sin+poly5+ridge	0.23623830289128636	0.9999839818067652

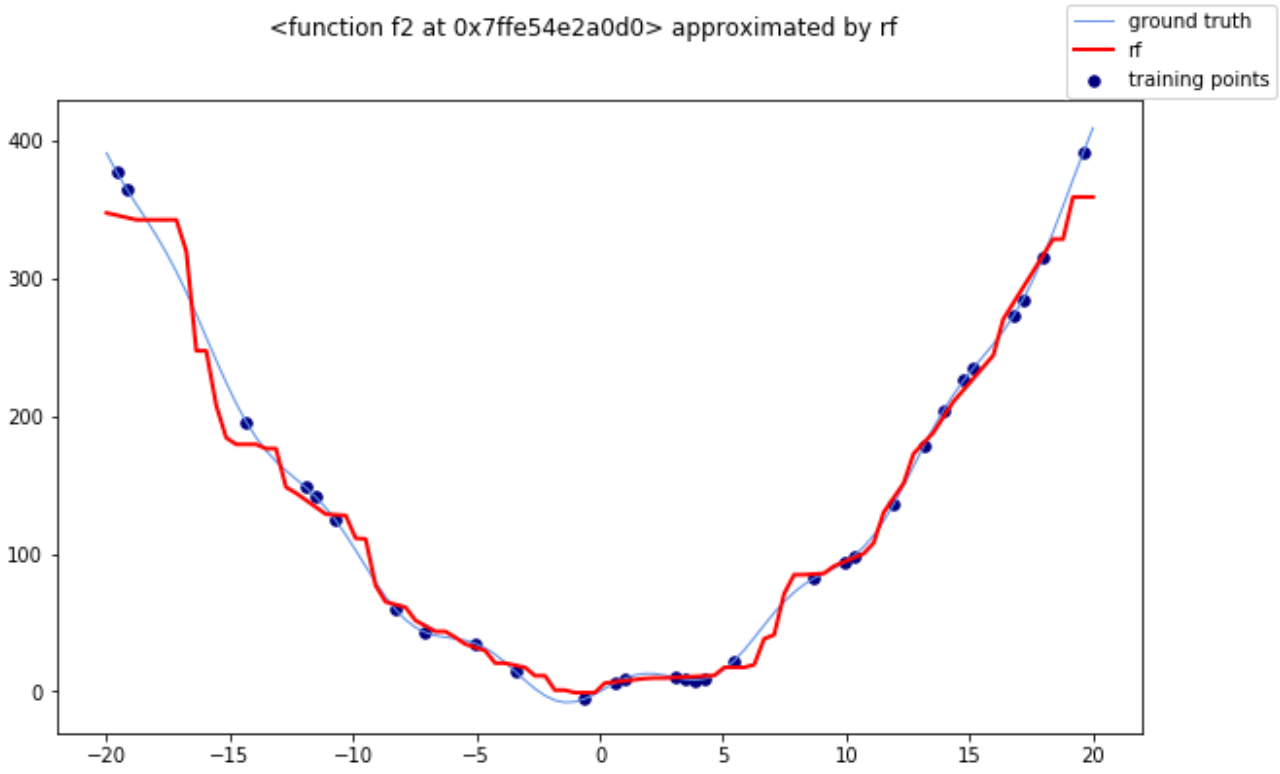


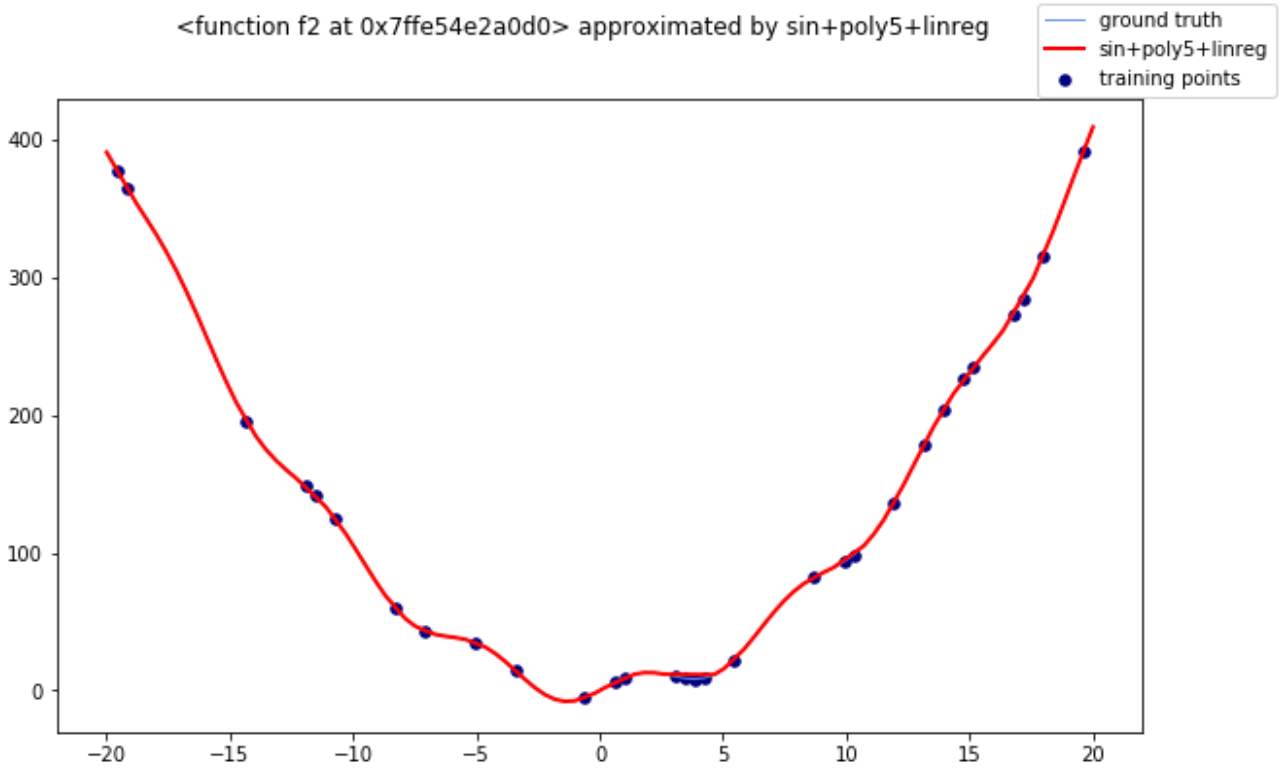


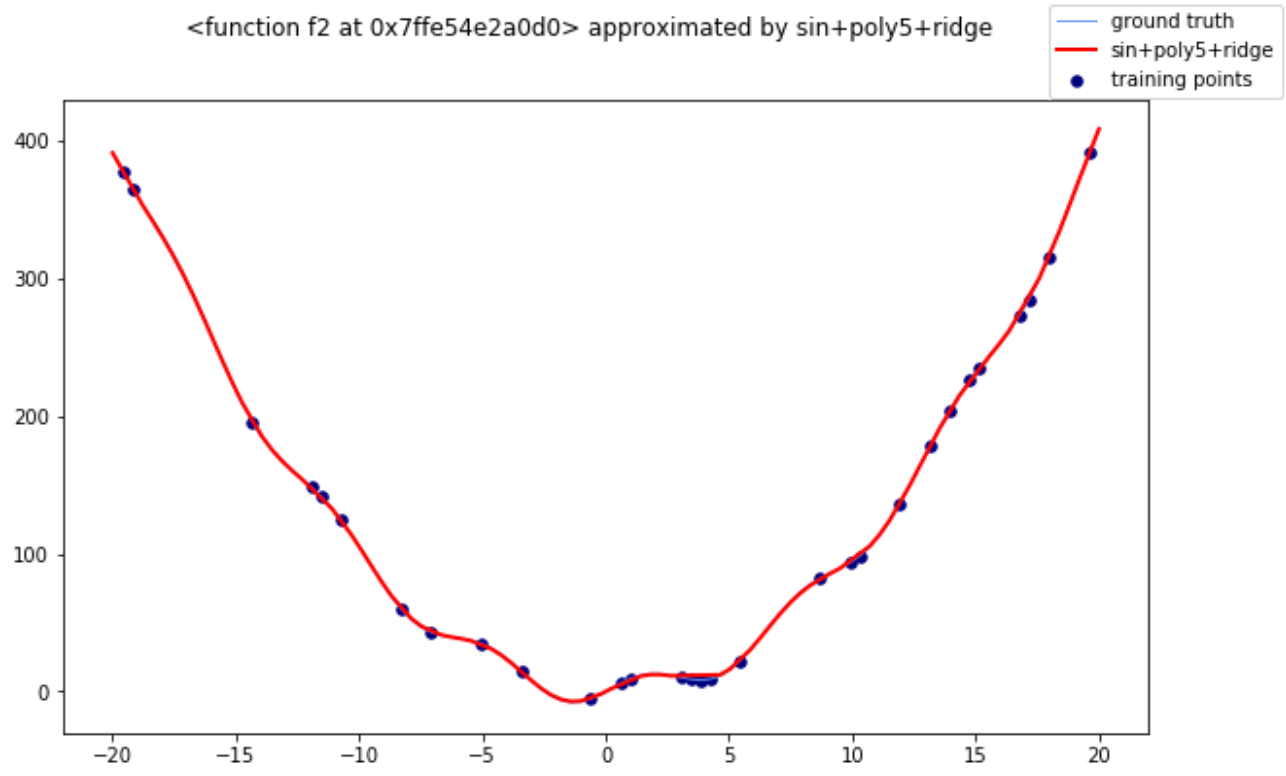












The crucial result here concerns the LinearRegression and Ridge models. Given the highly non-linear behavior, without a proper preprocessing, they performed the worst. However, the higher capability achieved including the sinusoidal component, along with polynomial features, drastically improved the performance. The best performing model here is the Ridge regularitazion in *sin+poly5+ridge* which copied almost identically the true values. This is a clear example of how an accurate preprocessing becomes crucial, especially for linear models.

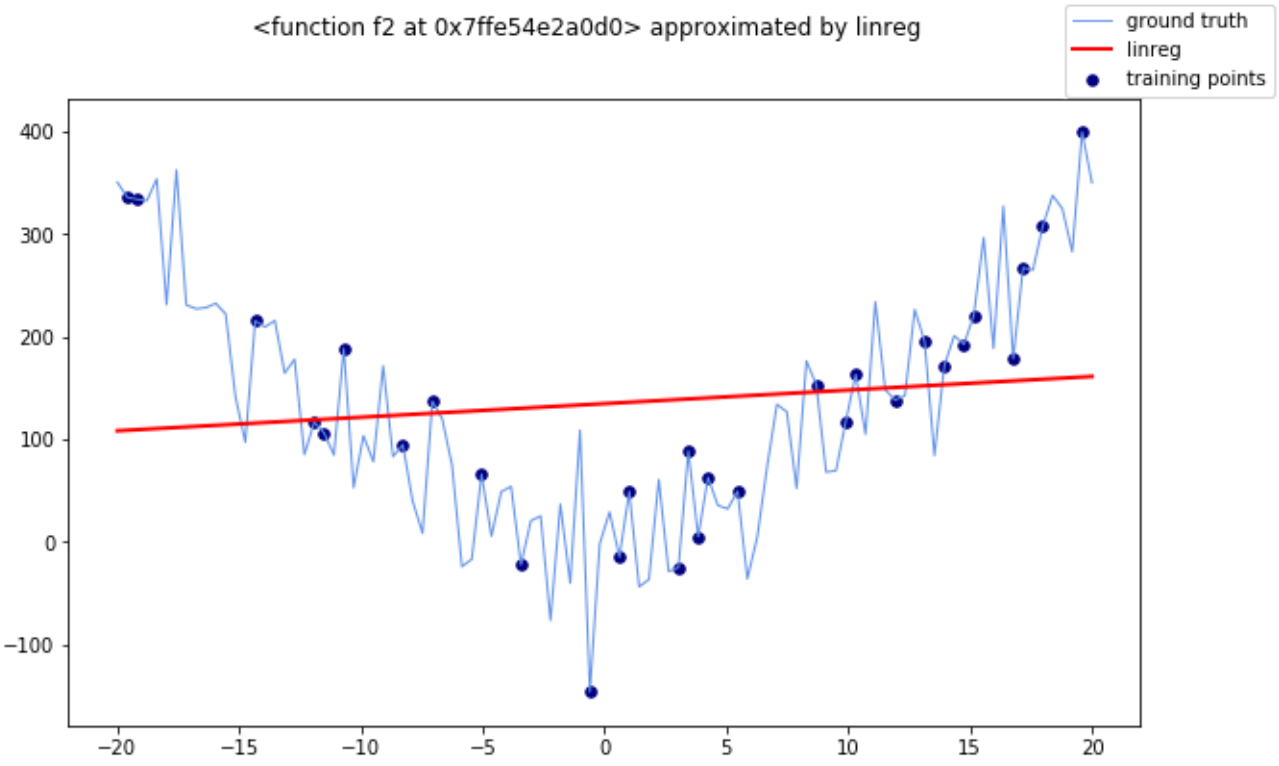
As a further exercise, we suggest to test the performance of the other regressors with the same preprocessing applied to improve the linear models. Let's test now what happens adding the Gaussian noise.

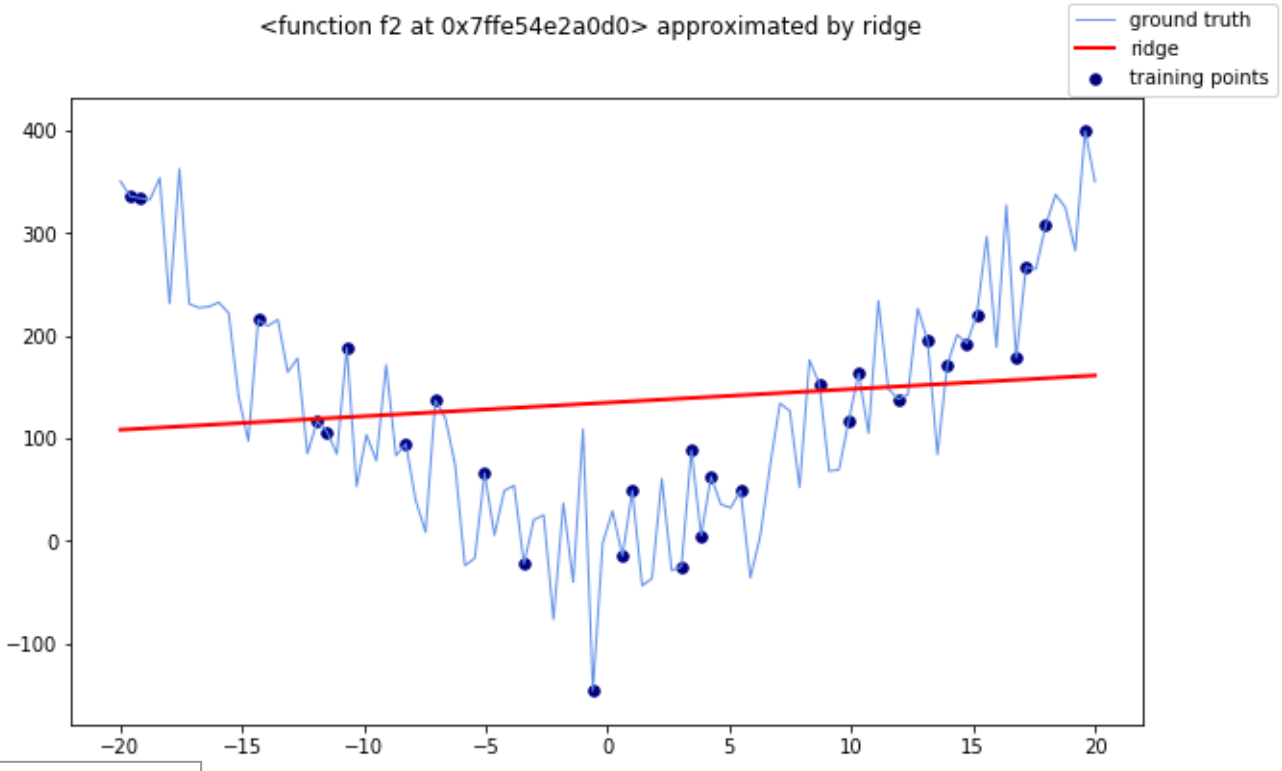
In [14]:

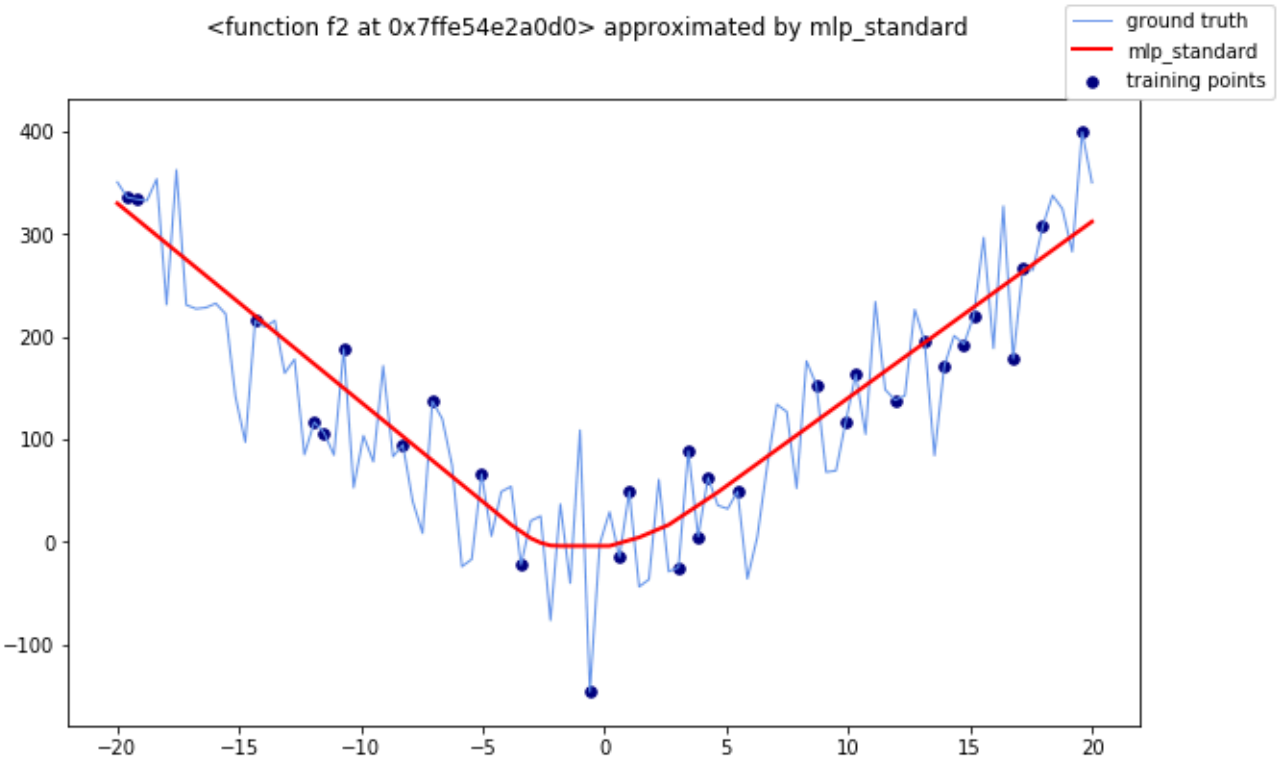
```
X, y = generate_X_y(f2)
y = inject_noise(y)

t = PrettyTable()
t.field_names = ['model', 'MSE', 'R2']
for model, name in zip(models, names):
    mse, r2 = evaluate_model(f2, X, y, model, name)
    t.add_row([name, mse, r2])

print(t)
```

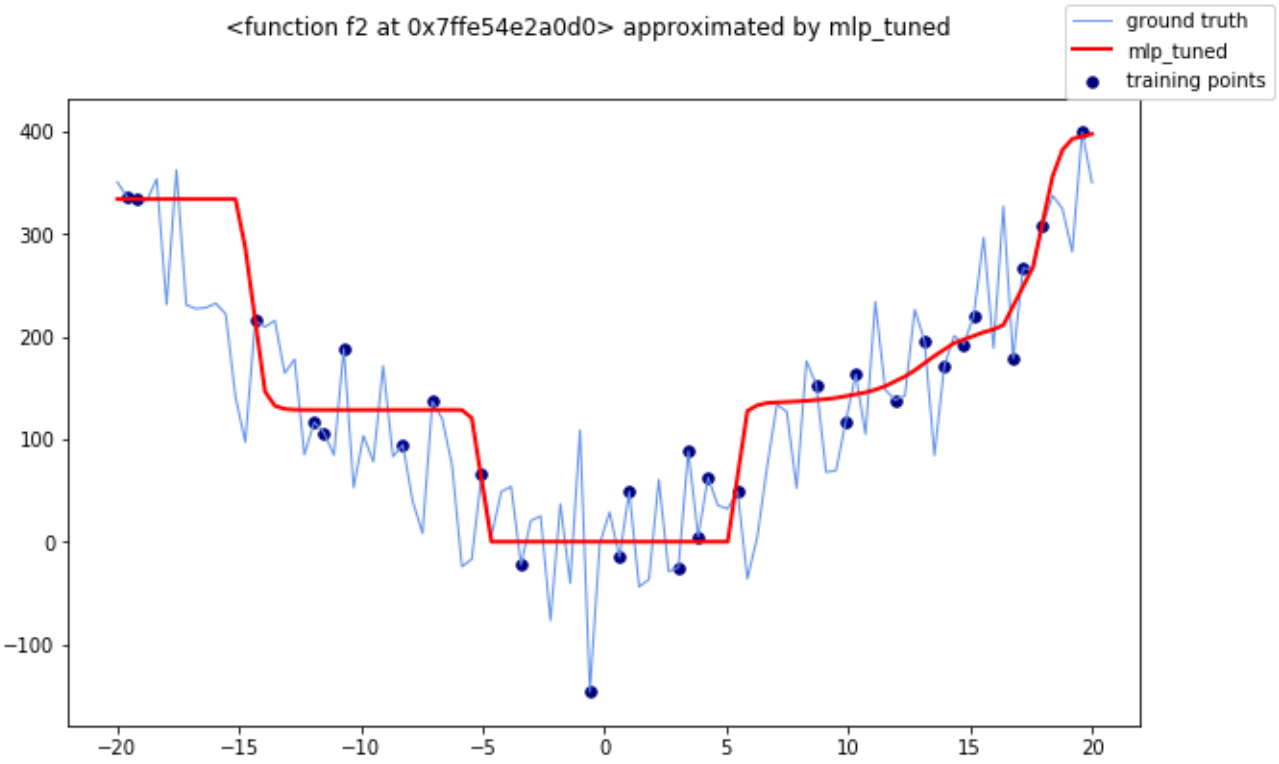
```
/Users/giuseppe/miniconda3/lib/python3.6/site-packages/sklearn/neural_networ
k/_multilayer_perceptron.py:571: ConvergenceWarning: Stochastic Optimizer: M
aximum iterations (10000) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
```

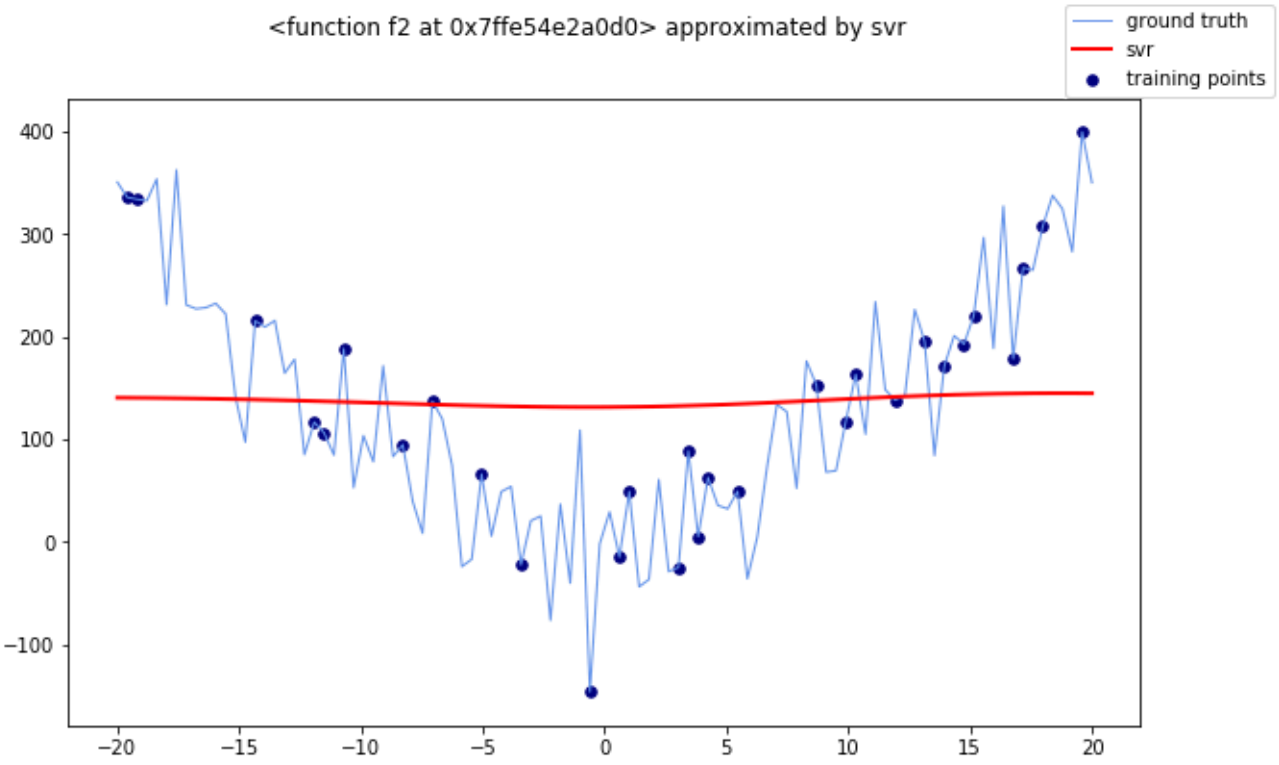
model	MSE	R2
linreg	13844.561167759284	-0.015583193249441907
ridge	13844.460508413049	-0.015575809271066499
mlp_standard	2991.1860609583377	0.7805782173532807
mlp_tuned	5787.52310195952	0.5754498014295486
svr	12891.137648265858	0.05435625016904311
rf	4113.044606937682	0.6982830351012688
sin+poly5+linreg	3314.773489446183	0.7568411013886113
sin+poly5+ridge	3298.91855356633	0.7580041578563832

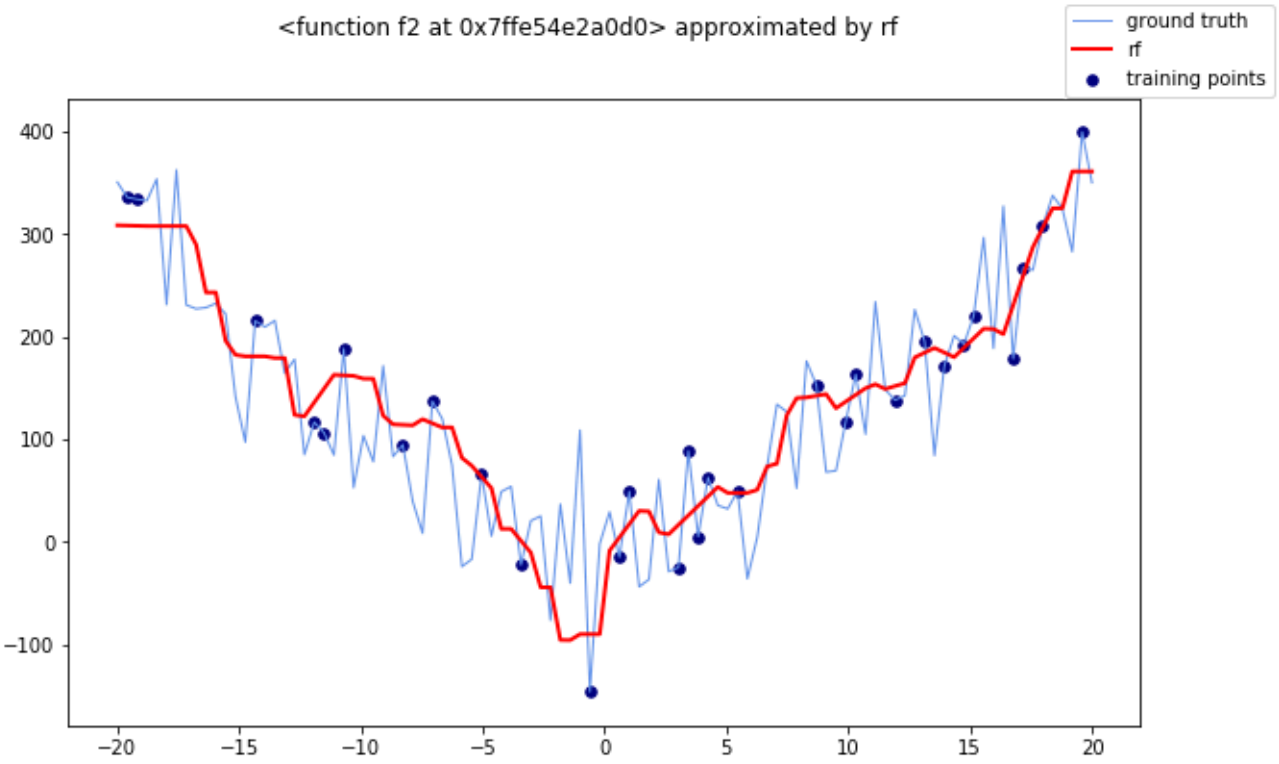


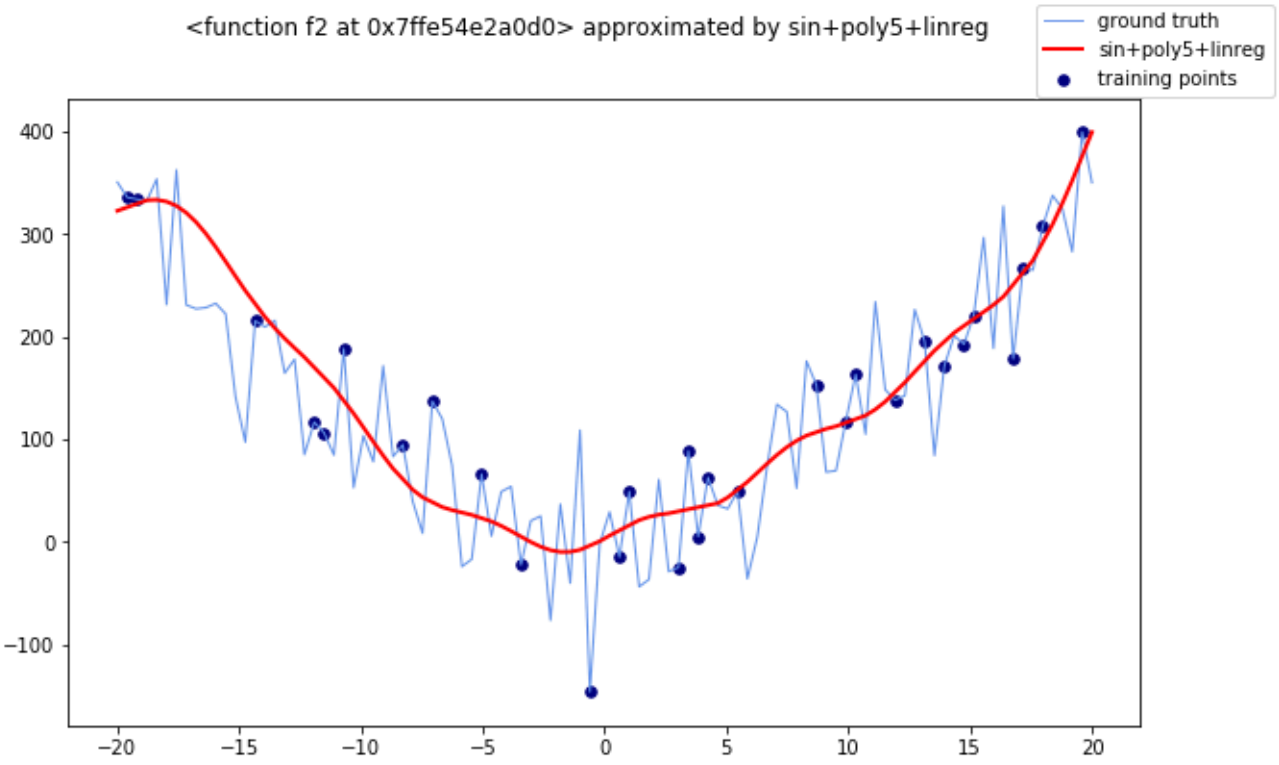


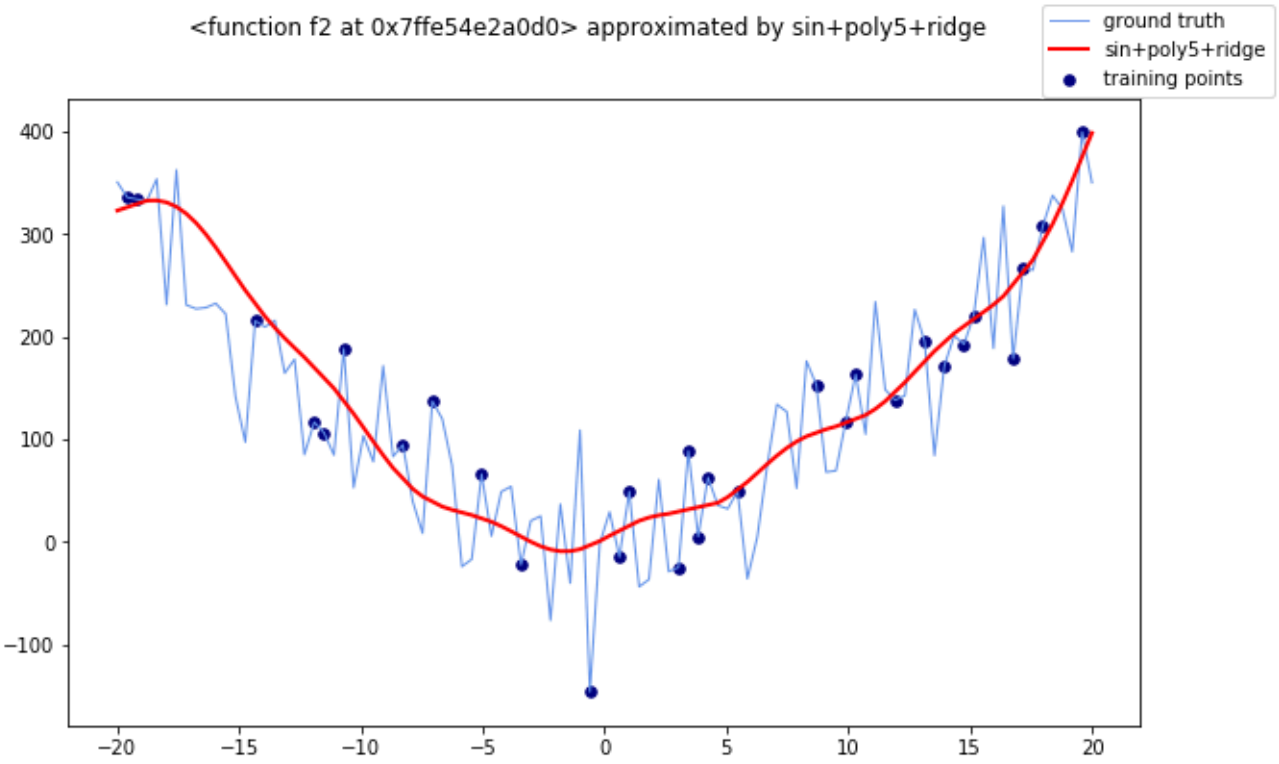












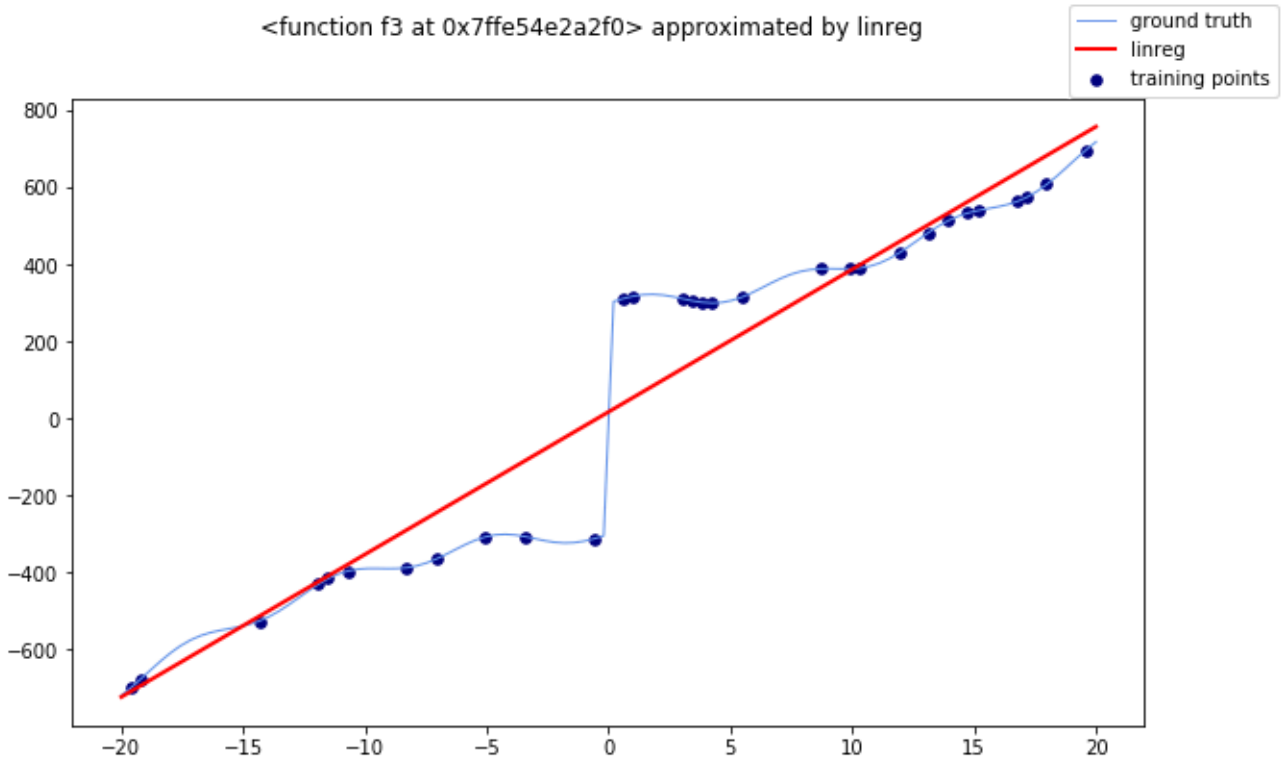
The overall shape of the function is retained by the standard MLP, and *sin+poly5+[linreg|ridge]*. This lead to the lowest overall error.

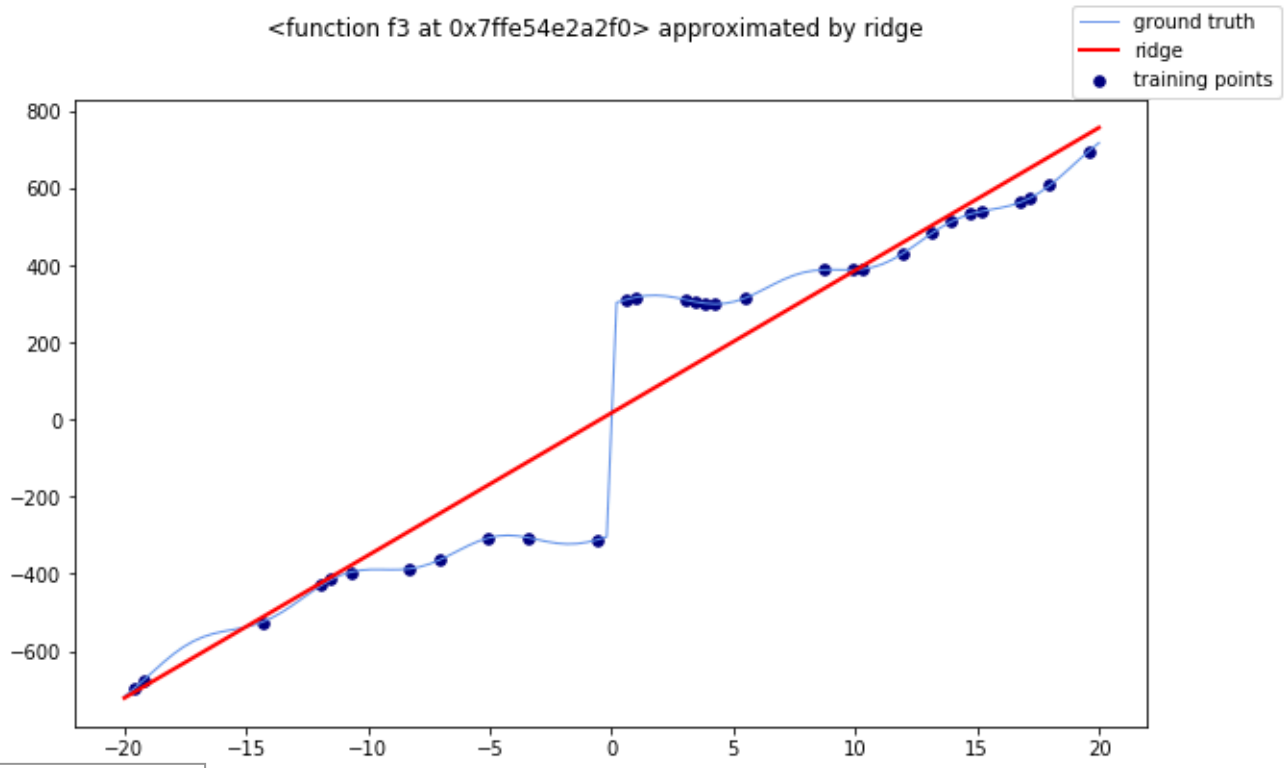
Results for f = f3

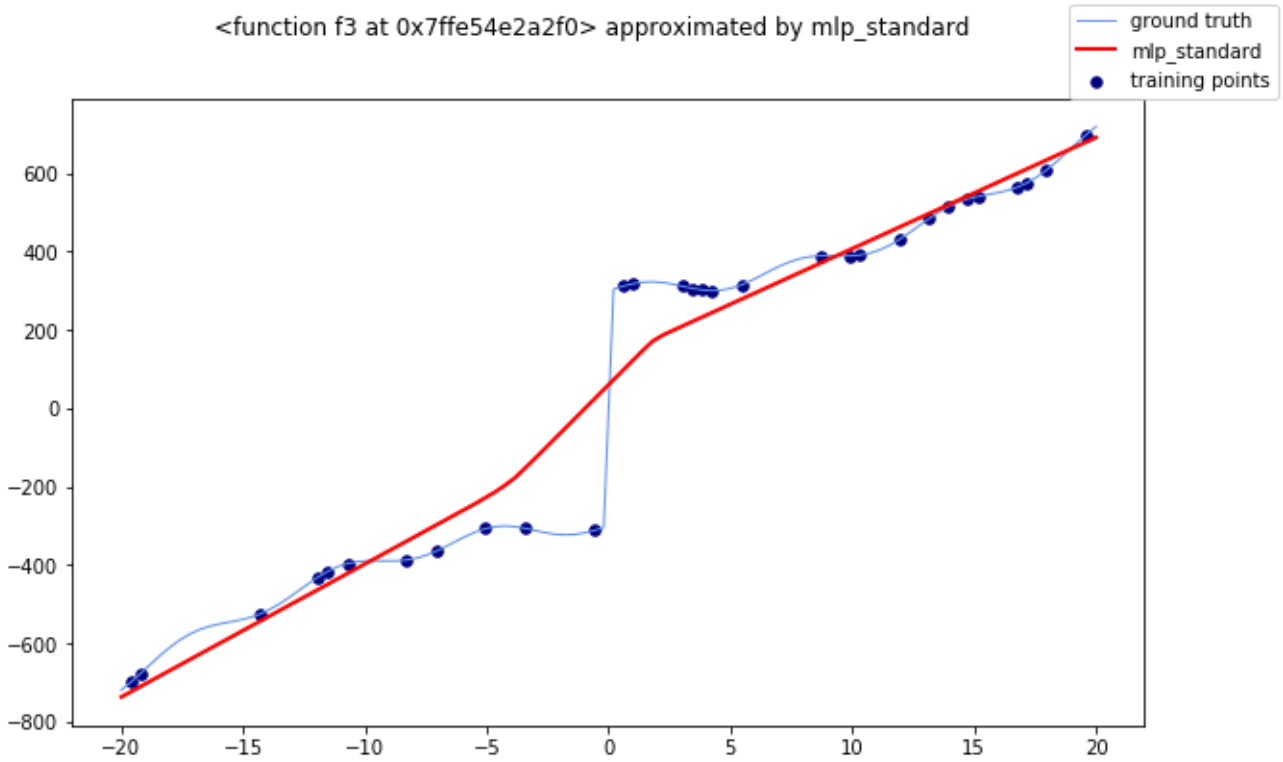
In [15]:
```
t = PrettyTable()
t.field_names = ['model', 'MSE', 'R2']
X, y = generate_X_y(f3)
for model, name in zip(models, names):
    mse, r2 = evaluate_model(f3, X, y, model, name)
    t.add_row([name, mse, r2])

print(t)
```

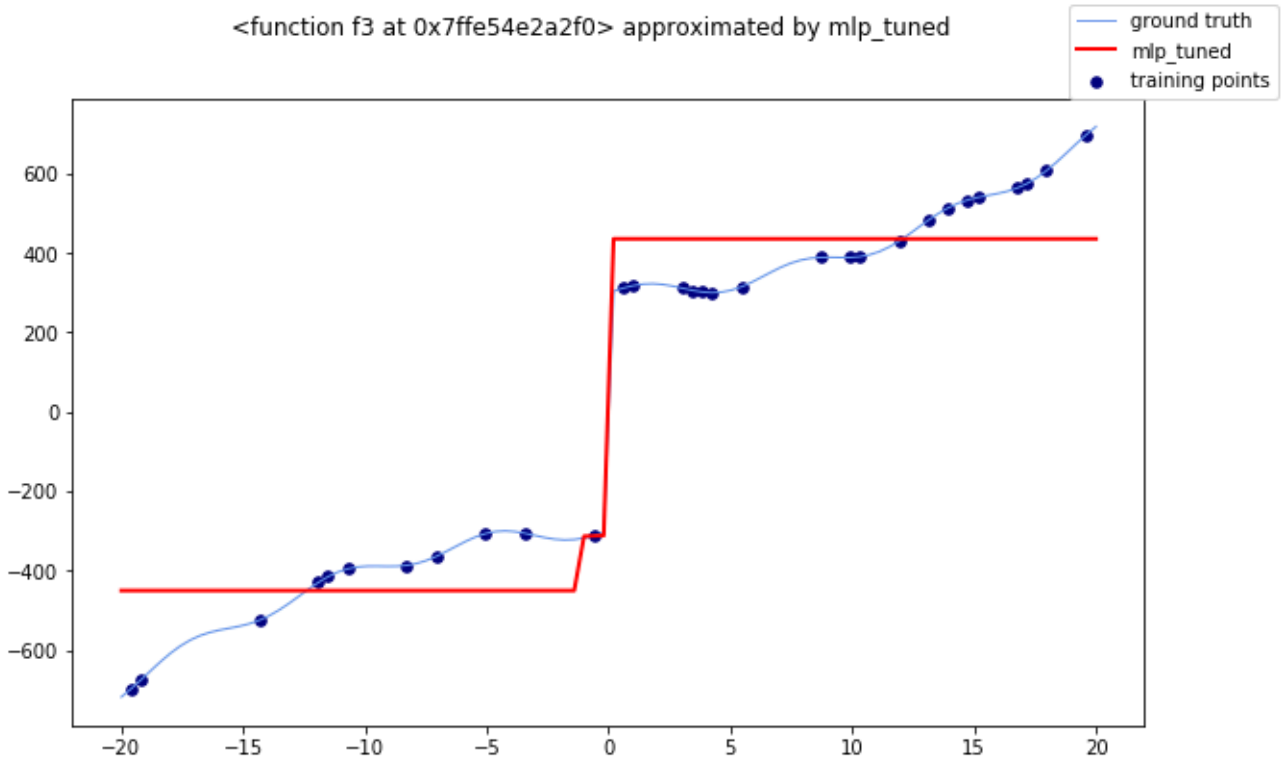
```
/Users/giuseppe/miniconda3/lib/python3.6/site-packages/sklearn/neural_networ
k/_multilayer_perceptron.py:571: ConvergenceWarning: Stochastic Optimizer: M
aximum iterations (10000) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
```

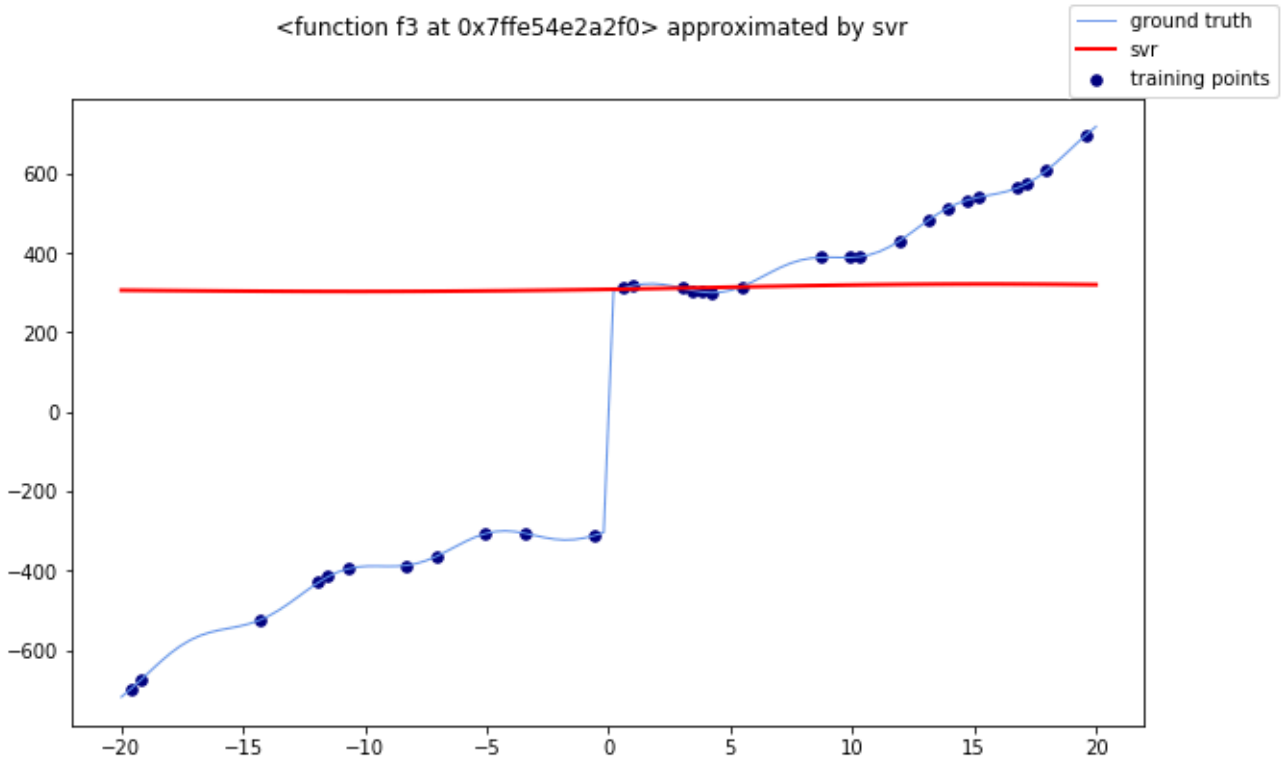
model	MSE	R2
linreg	15760.86079984127	0.9222471586304892
ridge	15763.614946346856	0.9222335716367915
mlp_standard	11102.721825876006	0.9452270926150279
mlp_tuned	13752.433310925047	0.9321553067913714
svr	326343.75456681015	-0.6099472295967054
rf	2451.5507620247454	0.9879057977905031
sin+poly5+linreg	9562.724944229945	0.9528243383980357
sin+poly5+ridge	9517.04484093962	0.9530496914336334

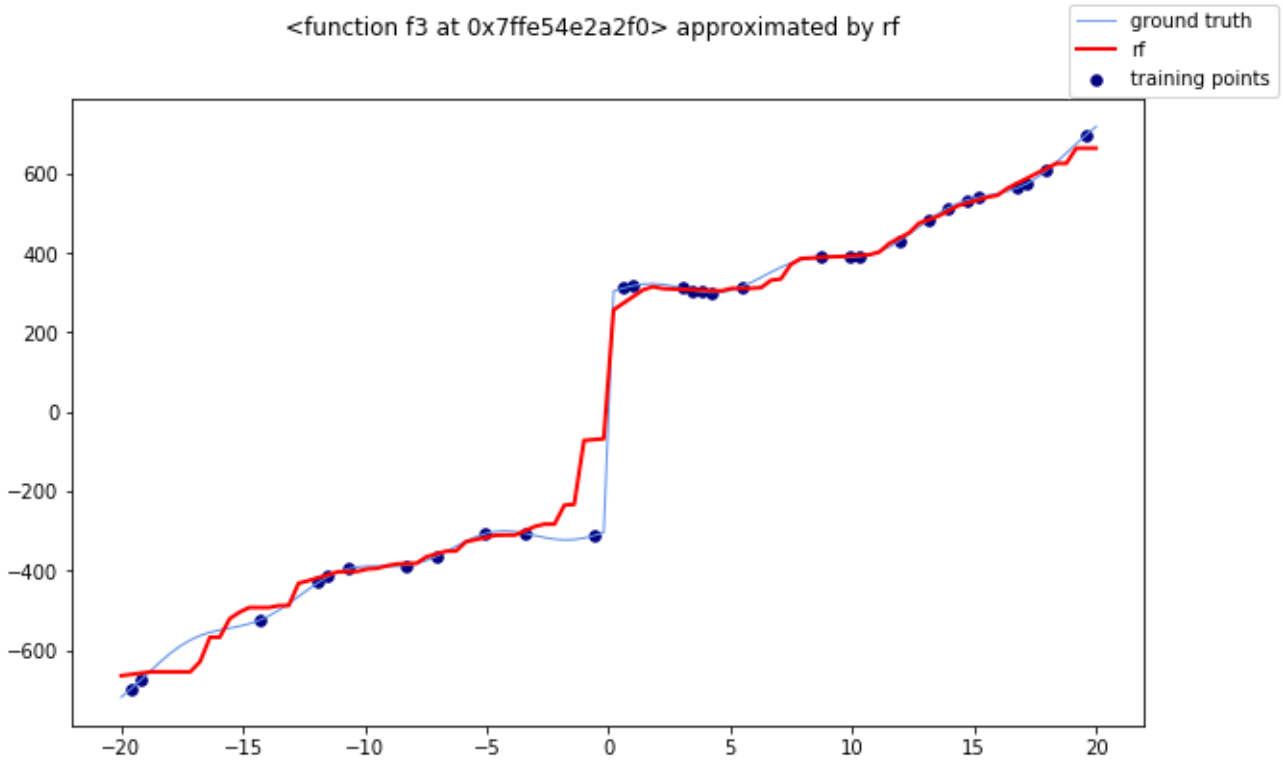


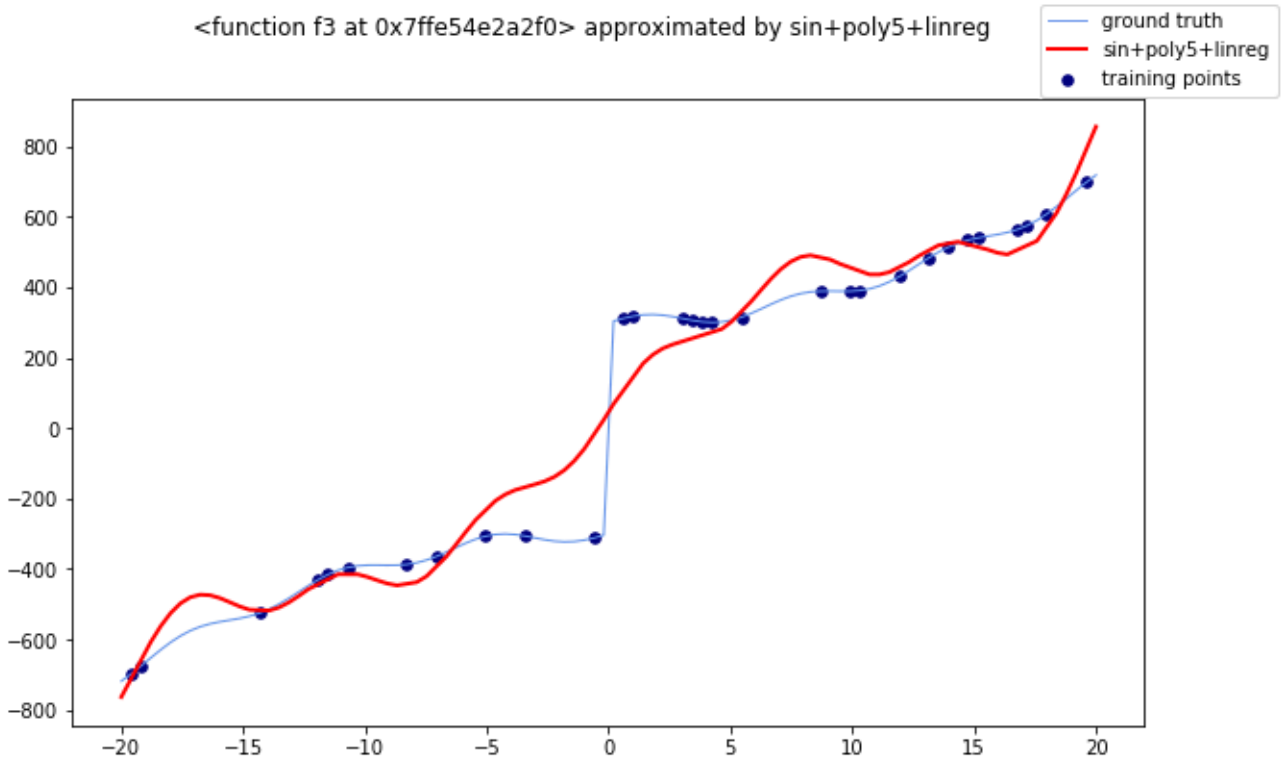


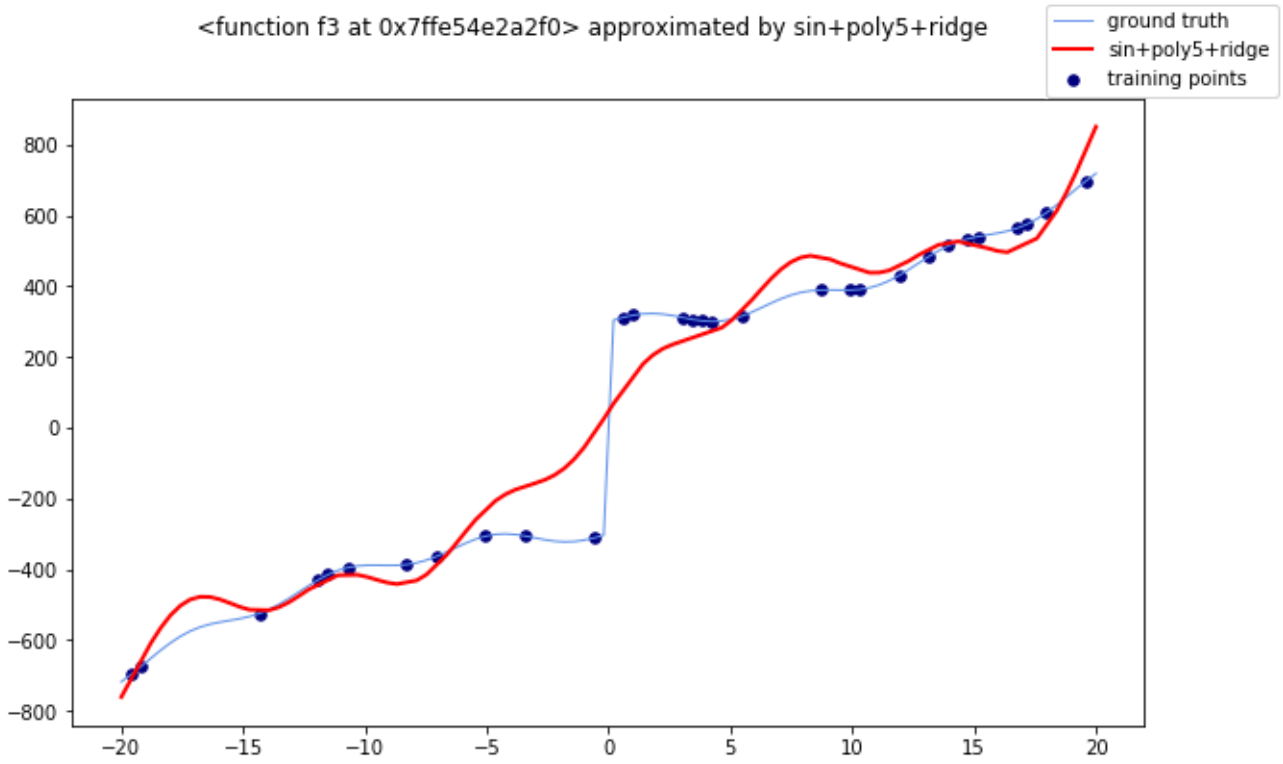












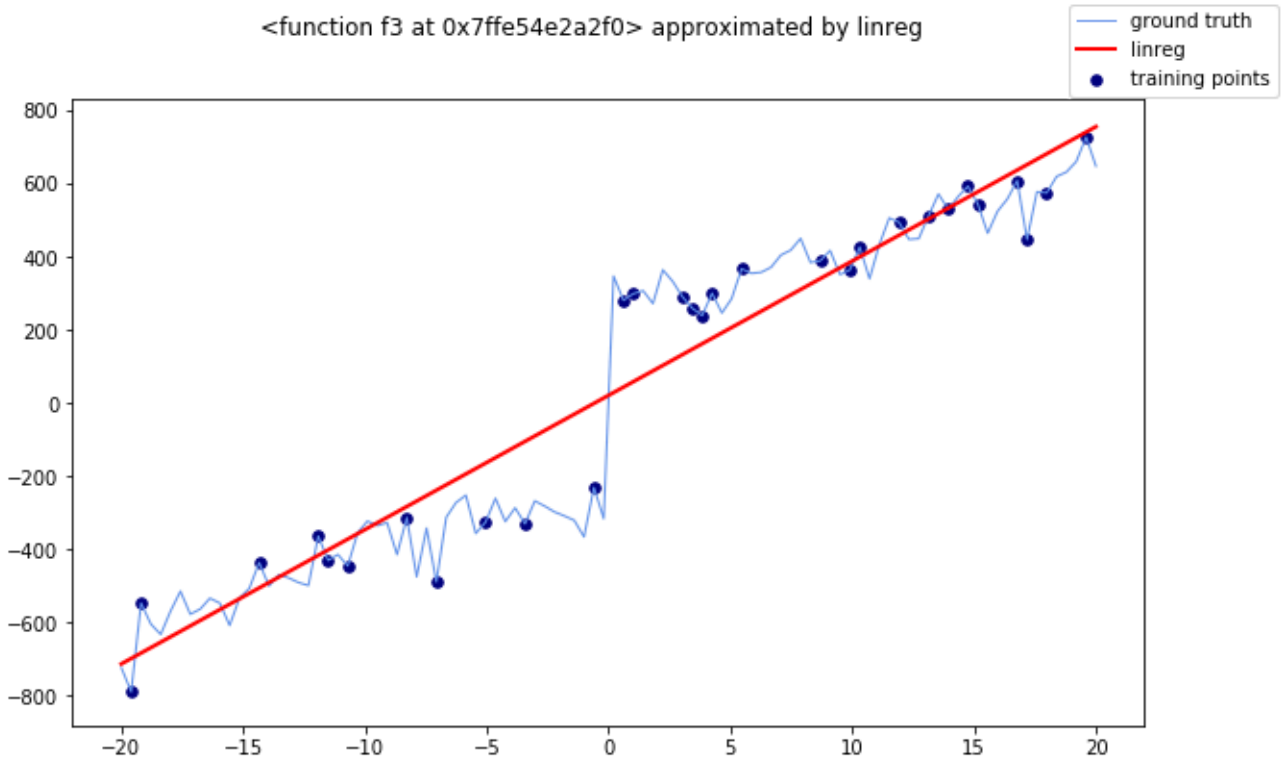
The top performing algorithm is Random Forest again. We can see that it is the only model able to handle the discontinuity at 0: values approaching 0 from the left (small negative values) are similar to the real values of the left-most part of the function, the same applies for small positive values.

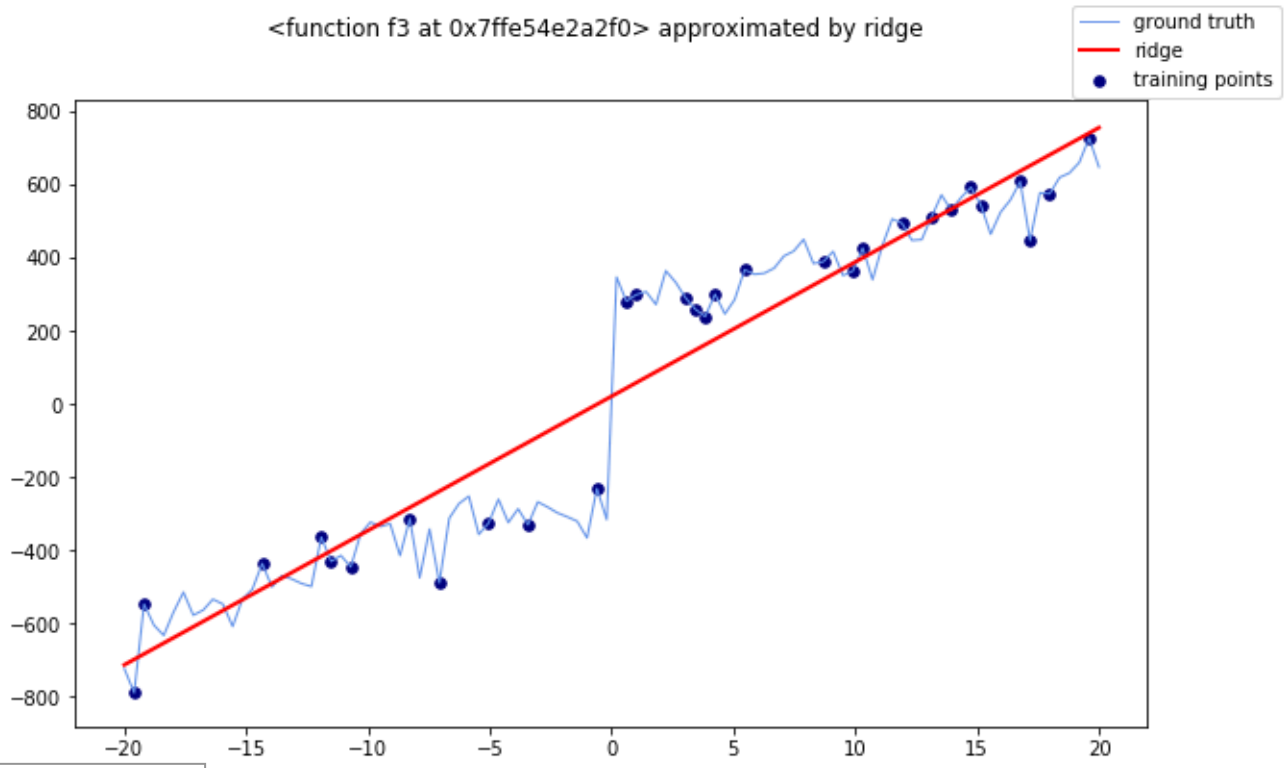
In [16]:

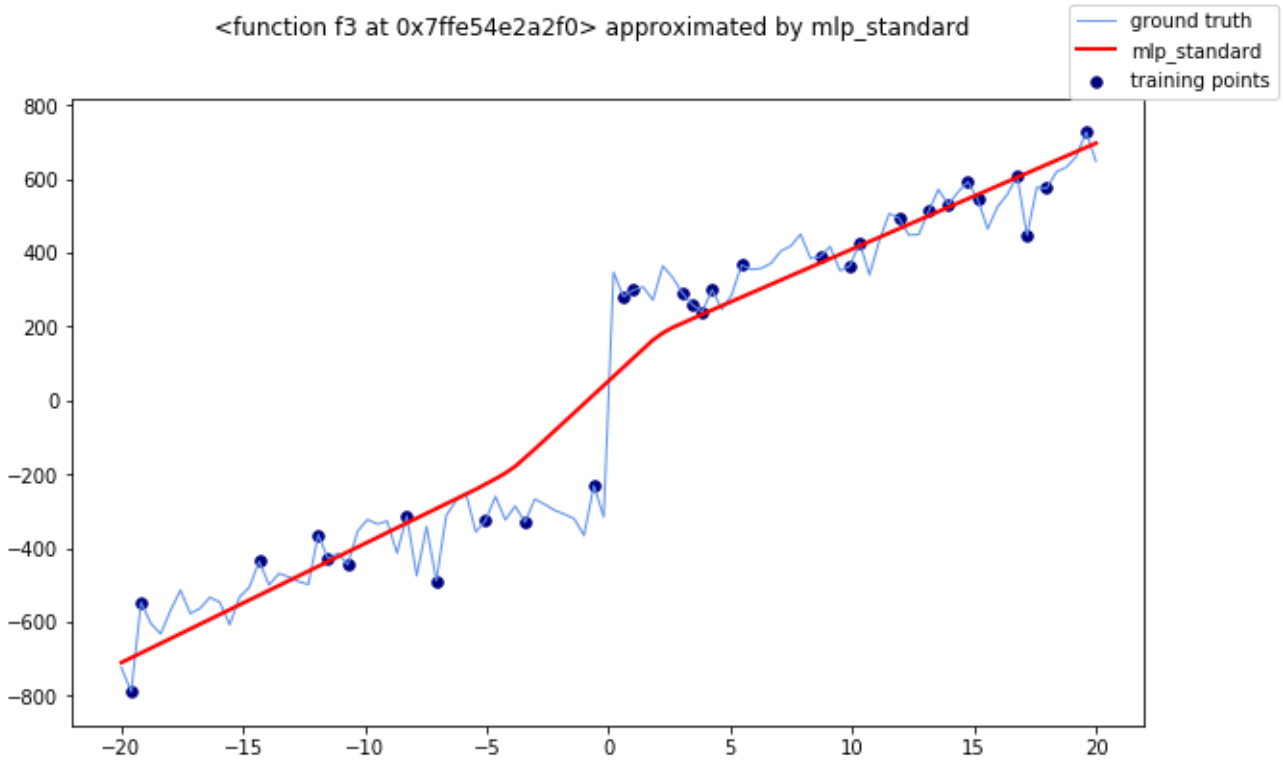
```
X, y = generate_X_y(f3)
y = inject_noise(y)

t = PrettyTable()
t.field_names = ['model', 'MSE', 'R2']
for model, name in zip(models, names):
    mse, r2 = evaluate_model(f3, X, y, model, name)
    t.add_row([name, mse, r2])

print(t)
```

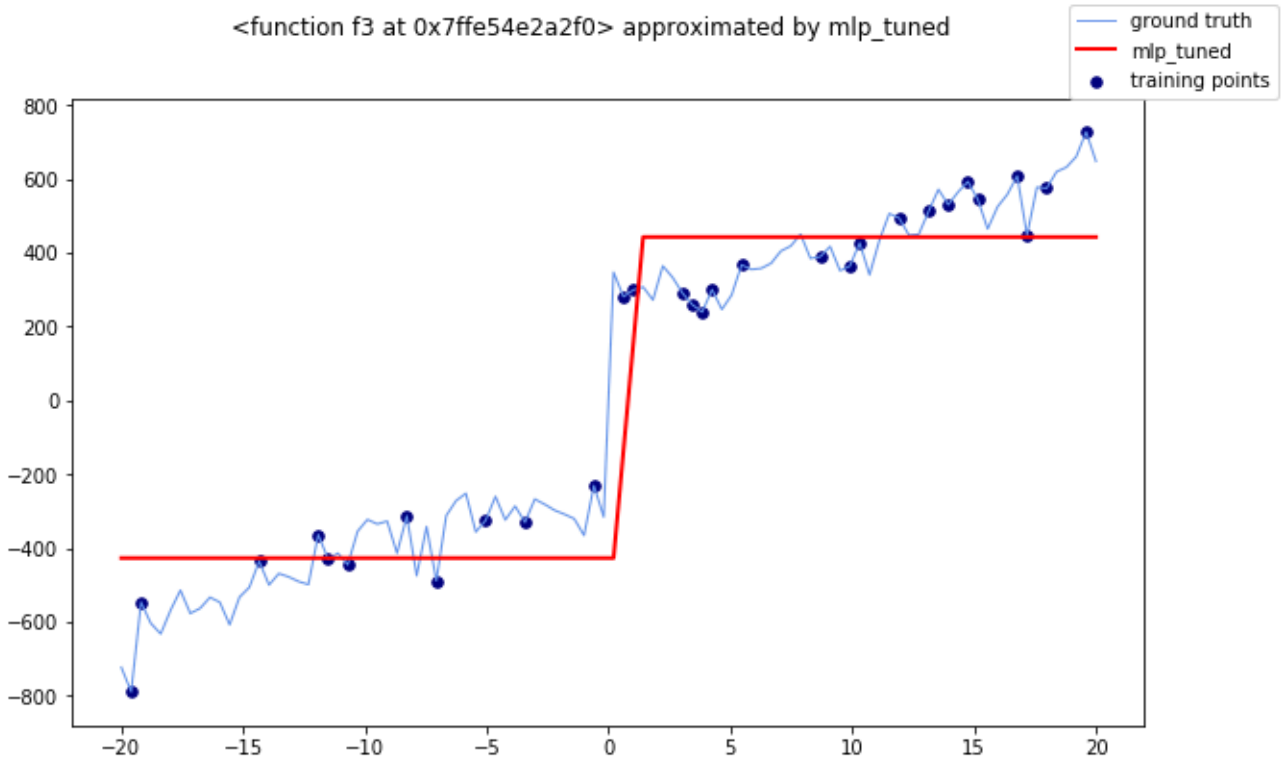
```
/Users/giuseppe/miniconda3/lib/python3.6/site-packages/sklearn/neural_network/_multilayer_perceptron.py:571: ConvergenceWarning: Stochastic Optimizer: Maximum iterations (10000) reached and the optimization hasn't converged yet.
  % self.max_iter, ConvergenceWarning)
```

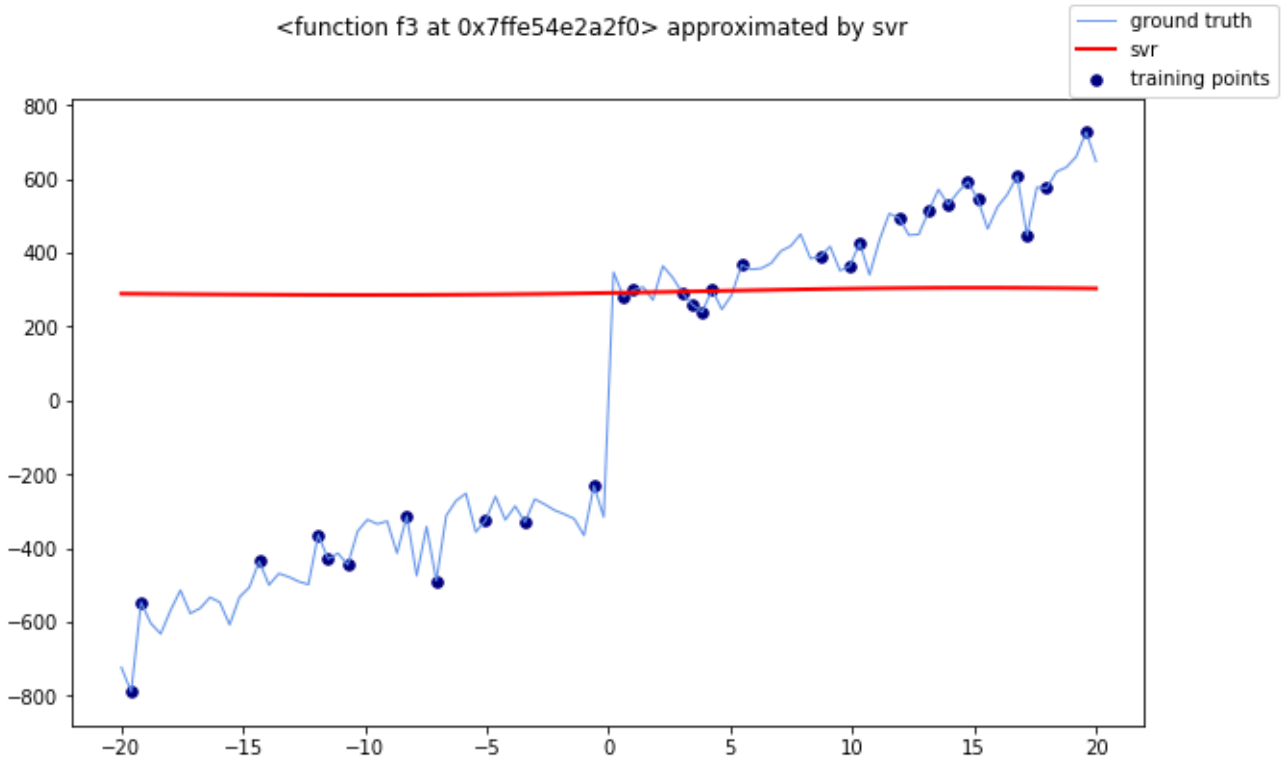
model	MSE	R2
linreg	17515.338908756312	0.9121094984003594
ridge	17517.105255078095	0.9121006350226635
mlp_standard	12352.939018232815	0.9380139880707941
mlp_tuned	22997.03282444203	0.8846028180914527
svr	305167.54329760035	-0.5313051372901239
rf	6353.001176701917	0.9681211729334968
sin+poly5+linreg	11793.78421643006	0.9408197799688728
sin+poly5+ridge	11650.803186975163	0.941537246782589

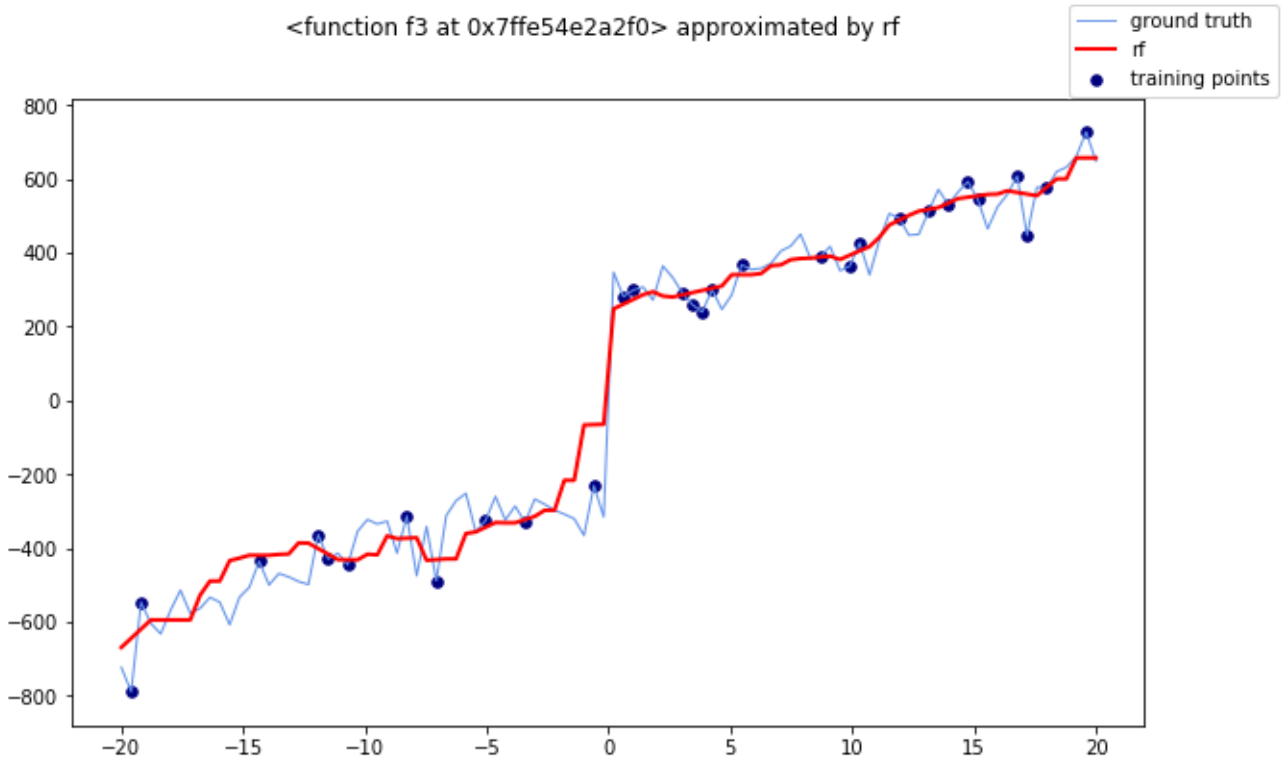


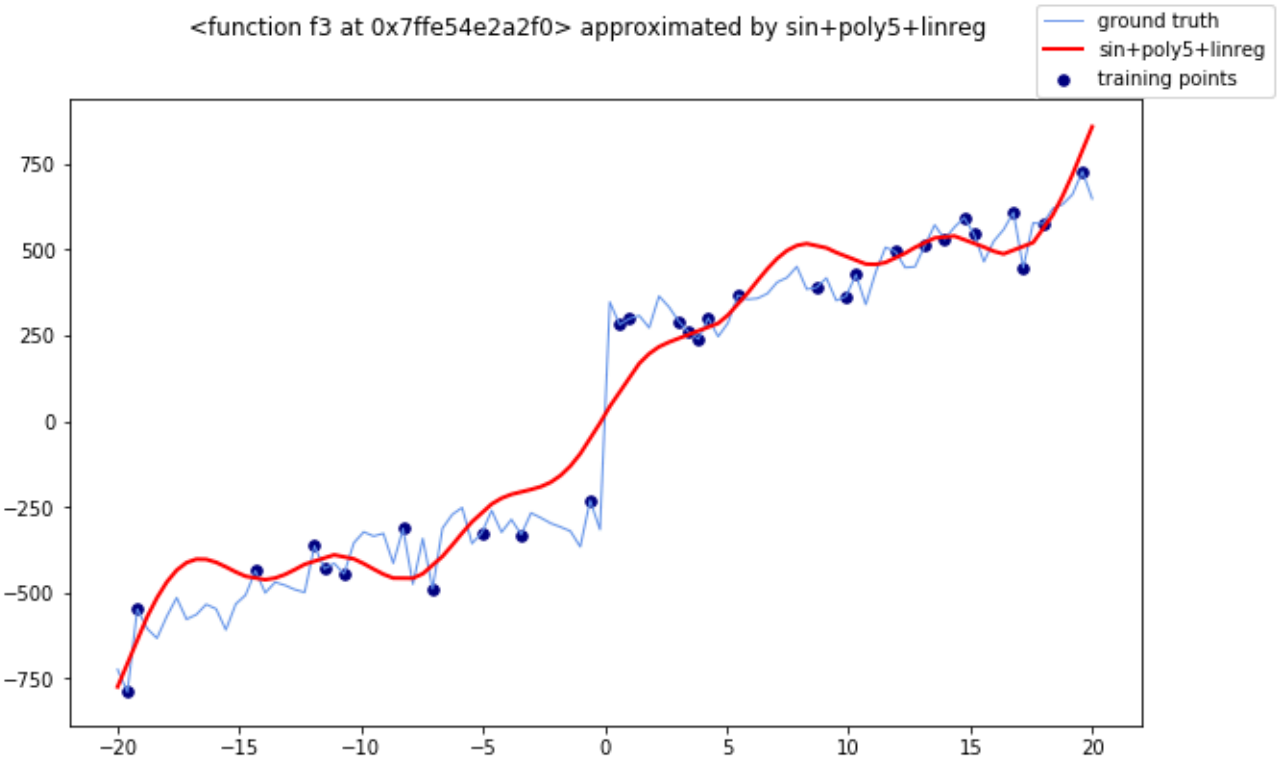


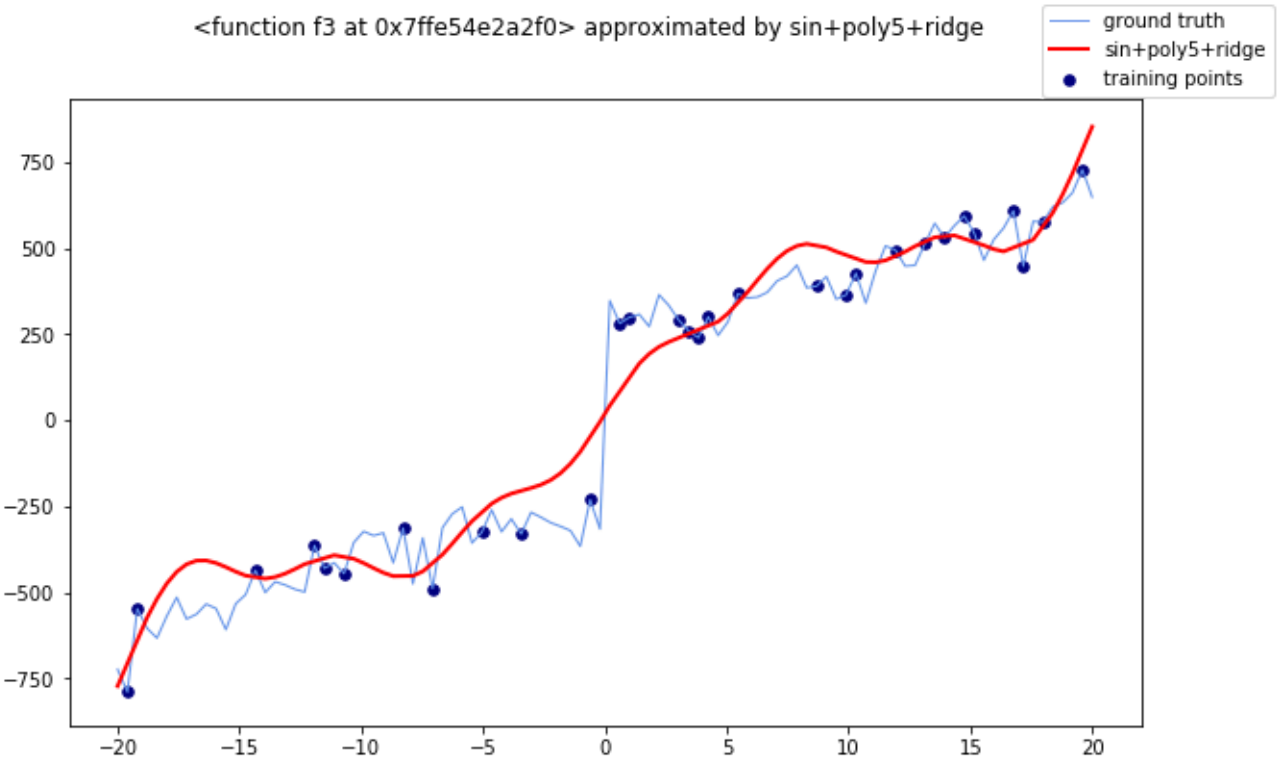












The most robust to noise is Random Forest in this case.

Exercise 2

In this exercise, you will carry out a multivariate regression analysis. Technically speaking, the preprocessing step added in the pipeline in Exercise 1 also lead to a multivariate analyis, considering also the newly generated features. Now, we generate a synthetic, multi-dimensional dataset using scikit-learn. The nature and the importance of each of the features can be fine-tuned using the [make_regression (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html)](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html) function.

In [17]:
```
from sklearn.datasets import make_regression
```

Exercise 2.1

We can use the `make_regression` function to generate a synthetic dataset with 2000 points. You should spend enough time inspecting the function parameters. For now, we recall that, by default:

- 100 features are generated, 10 of which are informative
- the target variable has a single dimension
- no noise is applied. You can set a normal noise with the parameter `noise`

In [18]:
```
X, y = make_regression(n_samples=2000, random_state=42)
X.shape, y.shape
```

Out[18]:
```
((2000, 100), (2000,))
```

Exercise 2.2

We can now run again the regression simulation developed in Exercise 1.

Note: here we get back to the normal conditions where we adopt 70% of the dataset as training set and the remaining 30% as test set. This can be achieved by simply changing the value to the `train_size` parameter.

In [19]:
```
t = PrettyTable()
t.field_names = ['model', 'MSE', 'R2']

for model, name in zip(models, names):
    X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                        train_size=.7,
                                                        random_state=42,
                                                        shuffle=True)
    model.fit(X_train, y_train)
    y_hat = model.predict(X_test)
    mse = mean_squared_error(y_test, y_hat)
    r2 = r2_score(y_test, y_hat)
    t.add_row([name, mse, r2])

print(t)
```

```
/Users/giuseppe/miniconda3/lib/python3.6/site-packages/sklearn/neural_networ
k/_multilayer_perceptron.py:470: ConvergenceWarning: lbfgs failed to converg
e (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  self.n_iter_ = _check_optimize_result("lbfgs", opt_res, self.max_iter)

+------------------+----------------------+----------------------+
|      model       |         MSE          |          R2          |
+------------------+----------------------+----------------------+
|      linreg      | 1.491861127842467e-25 |         1.0          |
|      ridge       |  0.026589221217723277 |  0.9999993361938854  |
|   mlp_standard   |   59.67097245363076   |  0.998510300243291   |
|    mlp_tuned     |   9081.738642192302   |  0.773272274785179   |
|       svr        |   38892.567627033684  |  0.029037970232504207 |
|        rf        |   10655.011193604842  |  0.7339951582793712  |
| sin+poly5+linreg |   40959.66218569419   | -0.022567527960504208 |
| sin+poly5+ridge  |    40620.5489939929   |  -0.01410148796816535 |
+------------------+----------------------+----------------------+
```

The tested model behave differently. Since the target label is generated by `make_regression` from a random linear model, we find that linear models and the model which can closely approximate a linear model (e.g. the standard MLP with one hidden layer) perform better. The R2 for the unregularized linear model is 1, meaning that it is able to capture all the information in the 10 informative features (the MSE is approximately 0).

Even though these results suggest that the so-defined problem is simple, we can notice that more complex models fail at grasping this simplicity, leading to high errors. Over-parametrized models strongly suffer the redundancy of information carried by the non-informative 90 features.

Exercise 2.3

Let's now inspect how the performance of our models change when:

- some noise is introduced;
- the number of informative features increases.

In [20]:
```
X, y = make_regression(n_samples=2000, random_state=42, noise=10)
t = PrettyTable()
t.field_names = ['model', 'MSE', 'R2']

for model, name in zip(models, names):
    X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                        train_size=.7,
                                                        random_state=42,
                                                        shuffle=True)
    model.fit(X_train, y_train)
    y_hat = model.predict(X_test)
    mse = mean_squared_error(y_test, y_hat)
    r2 = r2_score(y_test, y_hat)
    t.add_row([name, mse, r2])

print(t)
```

```
/Users/giuseppe/miniconda3/lib/python3.6/site-packages/sklearn/neural_networ
k/_multilayer_perceptron.py:470: ConvergenceWarning: lbfgs failed to converg
e (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  self.n_iter_ = _check_optimize_result("lbfgs", opt_res, self.max_iter)
```

model	MSE	R2
linreg	111.1154487123277	0.9972285317141203
ridge	111.32455388067977	0.9972233161626446
mlp_standard	397.79932377985136	0.9900779934493669
mlp_tuned	8652.930271788262	0.7841765289516479
svr	39028.249235400996	0.026548006935037116
rf	10049.773799052658	0.7493361212404551
sin+poly5+linreg	40425.61438644836	-0.008305411242192973
sin+poly5+ridge	40291.90586225675	-0.004970421025771943

As we might have expected, the introduction of a random gaussian noise with standard deviation of 10 decreases the performance of every model.

Let's introduce more informative features. However, keep in mind that those will be used to generate a target value with a linear combination: we can expect that models close to linear will be again the top performer.

In [21]:
```
X, y = make_regression(n_samples=2000, random_state=42, noise=10,
                       n_informative=70)
t = PrettyTable()
t.field_names = ['model', 'MSE', 'R2']

for model, name in zip(models, names):
    X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                        train_size=.7,
                                                        random_state=42,
                                                        shuffle=True)
    model.fit(X_train, y_train)
    y_hat = model.predict(X_test)
    mse = mean_squared_error(y_test, y_hat)
    r2 = r2_score(y_test, y_hat)
    t.add_row([name, mse, r2])

print(t)
```

```
/Users/giuseppe/miniconda3/lib/python3.6/site-packages/sklearn/neural_networ
k/_multilayer_perceptron.py:470: ConvergenceWarning: lbfgs failed to converg
e (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  self.n_iter_ = _check_optimize_result("lbfgs", opt_res, self.max_iter)

+-----------------+--------------------+----------------------+
|      model      |        MSE         |          R2          |
+-----------------+--------------------+----------------------+
|      linreg     | 99.92193369275083  |  0.9996581789295267  |
|      ridge      | 100.17526177647642 |  0.9996573123242324  |
|   mlp_standard  | 151.2964519724636  |  0.9994824328026811  |
|    mlp_tuned    | 70093.10550001937  |  0.7602198089111757  |
|       svr       | 288970.03486527514 | 0.011467822909904779 |
|        rf       | 193262.1739056526  |  0.3388730509404215  |
| sin+poly5+linreg | 287344.53105641843 | 0.01702847842834465  |
| sin+poly5+ridge | 287575.8006801805  | 0.016237332506315427 |
+-----------------+--------------------+----------------------+
```

Comparing the latter tables with the previous one (with 10 informative features) we can notice two different changes. While Linear Regression, Ridge and the standard MLP have improved their performance (in terms of both MSE and R2), the other, more complex classifiers performed significanlty worse. We can conclude that increasing the number of informative features not only brings more information (and lowers the redundancy), but also makes the problem harder, and the latter factor has the strongest impact on the tested models.

Exercise 2

In the second part of the laboratory, we will cover the topic of time series forecasting, working on the dataset of temperatures collected during the Second World War. Specifically, we will focus on the forecasting of the temperature value one - or more - day ahead of the current time, based on the history of available temperatures.

Exercise 2.1

To quickly load and inspect the dataset, we can use pandas.

In [22]:
```
import pandas as pd
```

In [23]:
```
df = pd.read_csv('weatherww2/SummaryofWeather.csv')
```

```
/Users/giuseppe/miniconda3/lib/python3.6/site-packages/IPython/core/interact
iveshell.py:3063: DtypeWarning: Columns (7,8,18,25) have mixed types.Specify
dtype option on import or set low_memory=False.
  interactivity=interactivity, compiler=compiler, result=result)
```

Exercise 2.2

We can inspect the content of the dataset with the method `info` .

In [24]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 119040 entries, 0 to 119039
Data columns (total 31 columns):
 #   Column       Non-Null Count   Dtype  
---  ------       --------------   -----  
 0   STA          119040 non-null  int64  
 1   Date         119040 non-null  object 
 2   Precip       119040 non-null  object 
 3   WindGustSpd  532 non-null     float64
 4   MaxTemp      119040 non-null  float64
 5   MinTemp      119040 non-null  float64
 6   MeanTemp     119040 non-null  float64
 7   Snowfall     117877 non-null  object 
 8   PoorWeather  34237 non-null   object 
 9   YR           119040 non-null  int64  
 10  MO           119040 non-null  int64  
 11  DA           119040 non-null  int64  
 12  PRCP         117108 non-null  object 
 13  DR           533 non-null     float64
 14  SPD          532 non-null     float64
 15  MAX          118566 non-null  float64
 16  MIN          118572 non-null  float64
 17  MEA          118542 non-null  float64
 18  SNF          117877 non-null  object 
 19  SND          5563 non-null    float64
 20  FT           0 non-null       float64
 21  FB           0 non-null       float64
 22  FTI          0 non-null       float64
 23  ITH          0 non-null       float64
 24  PGT          525 non-null     float64
 25  TSHDSBRSGF   34237 non-null   object 
 26  SD3          0 non-null       float64
 27  RHX          0 non-null       float64
 28  RHN          0 non-null       float64
 29  RVG          0 non-null       float64
 30  WTE          0 non-null       float64
dtypes: float64(20), int64(4), object(7)
memory usage: 28.2+ MB
```

Please refer to the laboratory text to discover more on the attributes.

The dataset is composed of 119040 rows. From the output of `info`, it is clear that most of the 31 attributes have missing values. 9 of them are completely null (e.g. FT, FB, etc.). This is commonplace in real-life tasks, where different sampling procedures are used during the time, the measurements can fail, etc.

The information relative to each sensor are reported in a second file. Let's read it.

In [25]:
```
sensors = pd.read_csv("weatherww2/WeatherStationLocations.csv")
sensors.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 161 entries, 0 to 160
Data columns (total 8 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   WBAN              161 non-null    int64  
 1   NAME              161 non-null    object 
 2   STATE/COUNTRY ID  161 non-null    object 
 3   LAT               161 non-null    object 
 4   LON               161 non-null    object 
 5   ELEV              161 non-null    int64  
 6   Latitude          161 non-null    float64
 7   Longitude         161 non-null    float64
dtypes: float64(2), int64(2), object(4)
memory usage: 10.2+ KB
```

The latter file contains useful information to characterize our sensors. Specifically, the Latitude and Longitude attribute can be used to group sensors near on the map. As we can see, many attributes in both the files are numerical, floating point. The normalization of these attributes is a matter specific to the considered task. Let's skip it for now, we will discuss about it later.

To discover the sensors with most of the temperature readings, we can use pandas. Note: this is not mandatory, every step in the following cells can be implemented with plain Python.

In [26]:
```
df.groupby("STA").size().sort_values(0, ascending=False).head(10)
```

Out[26]:
```
STA
22508    2192
10701    2185
22502    2154
22504    2118
10803    1750
11610    1631
16405    1622
11601    1604
10502    1527
11604    1514
dtype: int64
```

Grouping the rows by sensor id (STA), counting the size of each group, and, finally, sorting them gives us the list of STAs that show up the most.

Exercise 2.3

Transorming features with dates into Datetime objects is usually a good idea. Meanwhile we can set a new index for our dataset.

In [27]:
```
df["Date"] = pd.to_datetime(df["Date"])
df = df.set_index("Date")
```

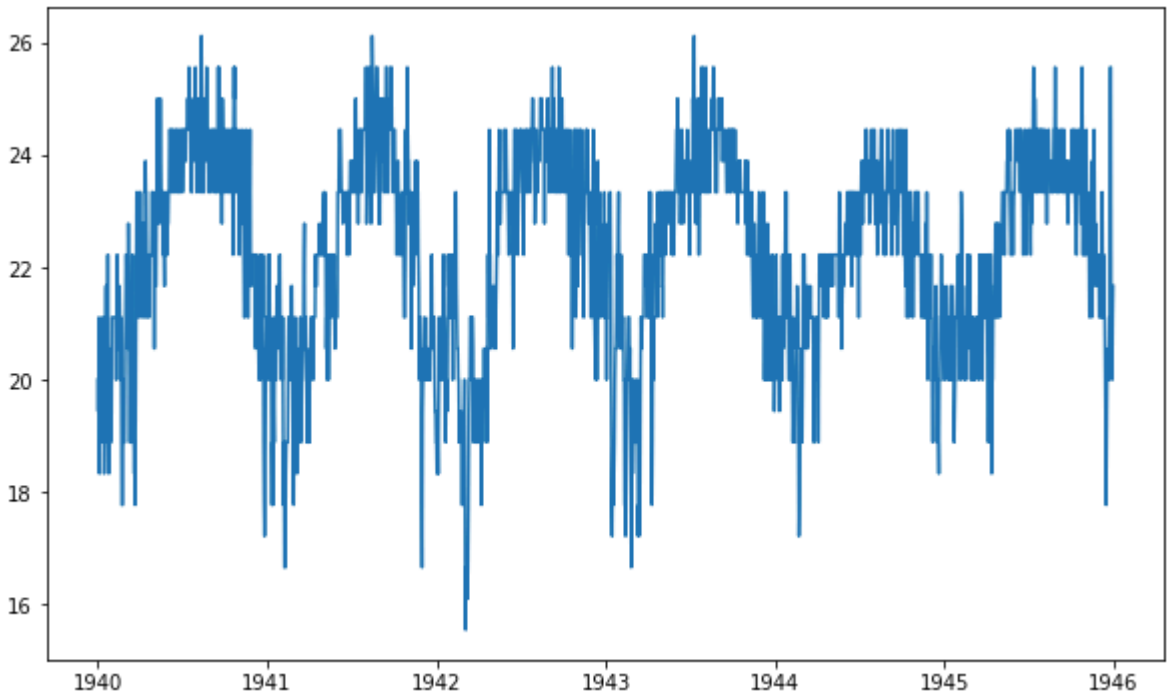
We can now keep only the readings of Sensor 22508 and plot the series of MeanTemps.

In [28]:
```
mtemps = df[df["STA"] == 22508]["MeanTemp"]
mtemps.head(5)
```

Out[28]:
```
Date
1940-01-01    20.000000
1940-01-02    19.444444
1940-01-03    20.000000
1940-01-04    21.111111
1940-01-05    18.333333
Name: MeanTemp, dtype: float64
```

Exercise 2.4

In [29]:
```
fig, ax = plt.subplots()
_ = ax.plot(mtemps.index.values, mtemps.values)
# or equivalently
# mtemps.plot(ax=ax)
```



This plot tells us that:

- the series has a clear seasonality. We may have expected that: the temperatures raise during the summer and fall down in winter;
- while the highest temperature of the year remain constant (except for the year 1944) in the period 1940-1946, the lowest one increases in the late years;
- given the observed range, we can suppose that the temperature values are expressed in Celsius degrees.

Exercise 2.5

There are many strategies to address the forecasting task of a time series. The transformation of the series itself into a structured representation (i.e. a set of records sharing some predictive features) enables the use of machine learning models. We also know these models are trained to learn the function mapping predicting features to a desired value. Thus, to complete the design of a forecasting framework, we need to carefully select a target variable.

ML practitioners often use the following strategy:

- predictive features: the features to be used in the structured representation are computed from the values already observed from the series. They can be the values themselves (which makes the approach more similar to classical, statistical autoregressive models like ARIMA (https://people.duke.edu/~rnau/411arim.htm)), or a more complex combination of them. In this exercise, we will adopt the simpler way and use the values in a window of fixed-length W. Specifically, this window will be rolling along the series, to obtain one record for each time step (for a graphical representation, please refer to the laboratory text);
- target variable: to model the task of forecasting, the target variable should encode a future event. Thus, the algorithm will be able to model the relationship between some already-seen feature values and an upcoming behavior of the series. In this exercise, we will use as target variable the value of the series right after the considered window.

In [30]:
```
W = 3
X = list()
y = list()
```

In [31]:
```
for i in range(mtemps.size - W): # range: [0, mtemps.size - W - 1]
    X.append(mtemps.iloc[i:i + W].values.T) # transpose to create a row arra
y
    y.append(mtemps.iloc[i + W])

# transform the structured representation into numpy arrays
X = np.array(X)
y = np.array(y)
X.shape, y.shape
```

Out[31]:
```
((2189, 3), (2189,))
```

We can check whether the rolling window has worked properly inspecting a few samples.

In [32]:
```
mtemps[:5]
```

Out[32]:
```
Date
1940-01-01    20.000000
1940-01-02    19.444444
1940-01-03    20.000000
1940-01-04    21.111111
1940-01-05    18.333333
Name: MeanTemp, dtype: float64
```

In [33]:
```
X[:3,:]
```

Out[33]:
```
array([[20.        , 19.44444444, 20.        ],
       [19.44444444, 20.        , 21.11111111],
       [20.        , 21.11111111, 18.33333333]])
```

In [34]:
```
y[:3]
```

Out[34]:
```
array([21.11111111, 18.33333333, 20.        ])
```

Given a record associated with the day *t*, `y[t]` is the value taken by the series at the day *t+W+1* (i.e. `mtemps[t+W]`), which seems correct.

Exercise 2.6

Let's use now the values from 1940 to 1944 as training data and test the model on the remaining year. We can leverage pandas to filter data based on time strings. To do so, we can convert our arrays into pandas DataFrame and Series using the DatetimeIndex from `mtemps`.

In [35]:
```
X_df = pd.DataFrame(X, index=mtemps.index[:mtemps.size - W],
                    columns=["t0", "t1", "t2"])
X_df.head()
```

Out[35]:

Date	t0	t1	t2
1940-01-01	20.000000	19.444444	20.000000
1940-01-02	19.444444	20.000000	21.111111
1940-01-03	20.000000	21.111111	18.333333
1940-01-04	21.111111	18.333333	20.000000
1940-01-05	18.333333	20.000000	20.555556

In [36]:
```
y_s = pd.Series(y, index=mtemps.index[:mtemps.size - W])
y_s.head()
```

Out[36]:
```
Date
1940-01-01    21.111111
1940-01-02    18.333333
1940-01-03    20.000000
1940-01-04    20.555556
1940-01-05    18.888889
dtype: float64
```

In [37]:
```
X_train, y_train = X_df.loc["1940":"1944"], y_s.loc["1940":"1944"]
X_train.index
```

Out[37]:
```
DatetimeIndex(['1940-01-01', '1940-01-02', '1940-01-03', '1940-01-04',
               '1940-01-05', '1940-01-06', '1940-01-07', '1940-01-08',
               '1940-01-09', '1940-01-10',
               ...
               '1944-12-22', '1944-12-23', '1944-12-24', '1944-12-25',
               '1944-12-26', '1944-12-27', '1944-12-28', '1944-12-29',
               '1944-12-30', '1944-12-31'],
              dtype='datetime64[ns]', name='Date', length=1827, freq=None)
```

Keep in mind that we used a forward-looking notation for our window, i.e. for the record *t* we have the values of the series between *t* and *t+W*. Under this conditions, at the end of the year, we are going to have *W* data points that include information from the next year. This is not a problem up if we slightly shift the beginning of our test set. Specifically, we might want to start our test set for the *Wth* day or the year.

Including some information from what it is consider the test set into the training procedure is a wrong habit known as **data leakage**. You should always stop and take a moment to consider if any data leakage is happening within your pipeline.

In [38]:
```
from datetime import date
initial_day = date(1944, 12, 31) + pd.Timedelta(f"{W} days")
initial_day
```

Out[38]:
```
datetime.date(1945, 1, 3)
```

In [39]:
```
X_test, y_test = X_df.loc[initial_day:], y_s.loc[initial_day:]
X_test.shape
```

Out[39]:
```
(360, 3)
```

Note how neat is the indexing on pandas object with Datetime indices. You are allowed to specify a numpy-like interval with dates as `datetime.date` or strings. Of course, [more complex and useful indexings are possibile (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html)](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html), but they go beyond the scope of the laboratory.

Exercise 2.7

Since we still have in memory the definitions of `models` and `names` (this is one of the beauties and curses of Jupyter notebooks) we can recycle and them for our current task.

In [40]:
```
t = PrettyTable()
t.field_names = ['model', 'MSE', 'R2']

for model, name in zip(models, names):
    model.fit(X_train, y_train)
    y_hat = model.predict(X_test)
    mse = mean_squared_error(y_test, y_hat)
    r2 = r2_score(y_test, y_hat)
    t.add_row([name, mse, r2])

print(t)
```

```
+-----------------+--------------------+--------------------+
|      model      |        MSE         |         R2         |
+-----------------+--------------------+--------------------+
|      linreg     | 0.7708838668711151 | 0.6684666582498298 |
|      ridge      | 0.7708190312576426 | 0.6684945420446823 |
|   mlp_standard  | 0.7756832982117083 | 0.6664025710646799 |
|    mlp_tuned    | 2.407530887487392  | -0.03540467610914  |
|       svr       | 0.7606632382515228 | 0.6728622349464781 |
|        rf       | 0.8362497037896549 | 0.6403548306695499 |
| sin+poly5+linreg | 1.1396245117527566 | 0.5098826957485592 |
| sin+poly5+ridge  | 1.1393130367062887 | 0.5100166515458784 |
+-----------------+--------------------+--------------------+
```

Even in this case, the models with behavior more close to linear perform the better. This could imply that some sort of linear relationship exists between the past `W` values of the series and the target. Additional comments can be made:

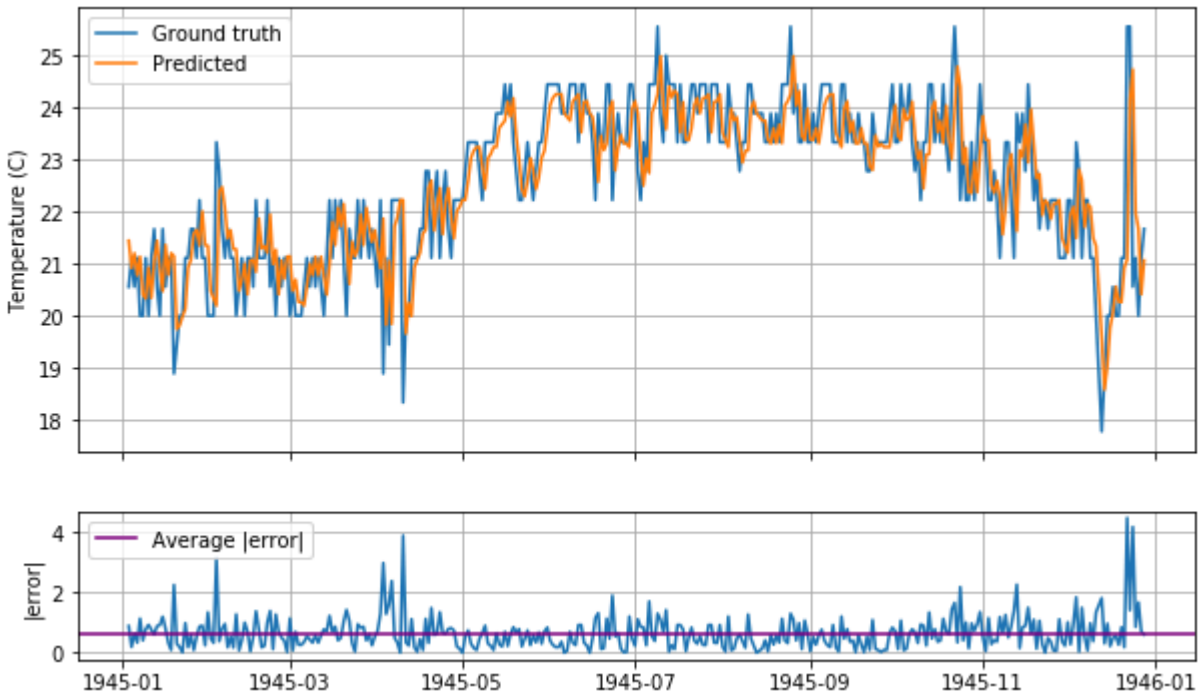
- the choice of custom tuning of MLP is wrong in this case, since it lead to the worst performance. Keep in mind that we do not have carried out any validation. The validation step is used for hyperparameter tuning, iterating the training process to identify the best configurations. However, validating a machine learning approach based on time series requires further attention. Cross-validation is not allowed, for example, since values have an intrinsc order and cannot be shuffled. Scikit-learn offers a validation strategy for time series thourgh the TimeSeriesSplit (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html) class. You can find complete examples on how to use it on the official documentation. The hyperparameter tuning can be mixed with TimeSeriesSplit the already seen GridSearchCV (http://scikit-learn.sourceforge.net/stable/modules/generated/sklearn.grid_search.GridSearchCV.html).
- our approach is intentionally non incremental, i.e. we train the model once and we use it to forecast an entire future year. Although this is correct for research purposes, in practical applications we might find useful to fed fresh information to the model (e.g. by running the training again) as it comes, while the time passes by and new records are collected. The profitability of this approach, however, depends on the application (e.g. there are cases in which you are not able to train the model iteratively in time) and an improvement in performance should not be taken for granted.

Exercise 2.8

Thanks to matplotlib, we can plot our predictions onto the real series for the test year. For simplicity, we can inspect the best performing model, the Ridge regularizer.

In [41]:
```
model = Ridge(random_state=42)
model.fit(X_train, y_train)
y_hat = model.predict(X_test)
y_hat = pd.Series(y_hat, index=y_test.index)
error = y_test - y_hat

fig, ax = plt.subplots(nrows=2, ncols=1, sharex=True, gridspec_kw={'height_ratios': [3, 1]})
ax[0].plot(y_test, label="Ground truth")
ax[0].plot(y_hat, label="Predicted")
ax[0].set_ylabel("Temperature (C)")
ax[0].legend()
ax[0].grid()
ax[1].plot(error.abs())
ax[1].set_ylabel("|error|")
ax[1].axhline(error.abs().mean(), color="purple", label = "Average |error|")
ax[1].legend()
ax[1].grid()

f"The average |error| is: {error.abs().mean():.2f} +- {error.abs().std():.2f} degrees Celsius"
```

Out[41]:
```
'The average |error| is: 0.65 +- 0.59 degrees Celsius'
```



The figure shows that Ridge provides a good approximation but its predictions are somewhat lagged. The model takes at least one day to react to the current trend and so induces a non-zero error. On the bottom-end of the figure, the plot of residuals shows that the larger is the change in temperature w.r.t. the previous day (i.e. a high volatility among consecutive days is present), the larger is the error due to the lagged limitations of the estimator.

Even though these results suggest that the only values of the series are not sufficient to build a robust forecasting model (e.g. we may think that other, more expressive features are needed), the average absolute error 0.65 and the standard deviation is 0.59, which may be acceptable values for our domain after all.

Predicting a value further than one day ahead is not possibile at the moment, but many strategies exist to expand the forecast horizon. For example, one can train different models on different target variables: one encoding the value one day ahead (i.e. what we have done here), one with the value two days ahead as target, and so on. In this experimental setting, you would end up having one model per horizon step, which could add some computational cost.

Exercise 2.9

As a further exercise, we leave to you the tweaking of this pipeline to test and visually assess the performance of the remaining regressors.