In [1]:
```
# setup
from mlwpy import *
diabetes = datasets.load_diabetes()
%matplotlib inline
```

Evaluating why less is more

The biggest risk in developing a learning system is overestimating how well it will do when we use it, What we can o protect ourselves from overconfidence? Our most fundamental defense is `not teaching to the test` To avoid teaching to the test, we use a very practical three-step recipe:

- **Step one**: split our data into separate training and testing datasets.
- **Step two**: learn on the training data.
- **Step three**: evaluate on the testing data.

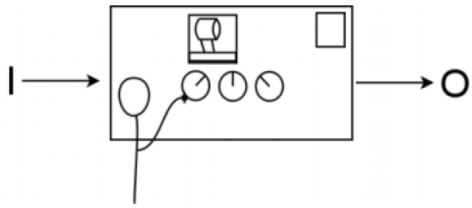
Some people might disagree and say that training with all of the data is better for the model, and it should but ***How would we know that a more-data model is better than a less-data model?***

Terminology for Learning Phases

We've hit on training and testing earlier. I want to introduce another phase called validation.

Back to the Machines

We will consider our machine learning model as a physical machine with a tray to feed the input to, output from the machine and a number of knobs to manipulate the input or the process



There is a business that want to use our machine so we need to build the machine, set all the knobs as in and send the machine to the customer. They won't do anything other than feed it inputs and see what pops out the other side.
Our challenge is to ensure the machine can perform adequately after the hand-off.

What could go wrong? There are many different types of machines that relate inputs to outputs. We've already seen two classifiers and two regressors. One example is the `K-NN` classifier .
For k-NN, different values of k are entirely different physical machines. k is not a knob we adjust on the machine so we could have a 3-NN or 10-NN or more so what we would do

Remember that we always set aside some data that the machine hasn't seen before for testing, we could use this data to pick the best machine ***BUT*** There is a problem, let's say we used this dataset to pick the best `k` for the model, What happen when it's time to test the model, The machine has already seen the data
We'll have two sets of held-out data to deal with two separate steps. One set will be used to *pick the machine*. The second set will be used to *evaluate how well the machine will work for the customer in a fair manner*

More Technically Speaking . . .

We now have three distinct sets of data. We can break the discussion of our needs into three distinct phases, We'll work from our final goal towards fitting the basic models.

1. We need to provide a single, well-tuned machine to our customer. We want to have a final, no-peeking evaluation of how that machine will do for our customer.
2. After applying some thought to the problem, we select a few candidate machines. With those candidate machines, we want to evaluate and compare them without peeking at the data we will use for our final evaluation.
3. For each of the candidate machines, we need to set the knobs to their best possible settings. We want to do that without peeking at either of the datasets used for the other phases. Once we select one machine, we are back to the basic learning step: we need to set its knobs.

Learning Phases and Training Sets

Each of our phases use a different dataset, let's give the phases some names:

1. **Assessment**: final, last-chance estimate of how the machine will do when operating in the wild
2. **Selection**: evaluating and comparing different machines which may represent the same broad type of machine (different k in k-NN) or completely different machines (k-NN and Naive Bayes)

3. **Training**: setting knobs to their optimal values and providing auxiliary side-tray information

The datasets used for these phases are:

1. Hold-out test set (HOT)
2. Validation test set (ValS)
3. Training set

One important note to remember that we can only use HOT once otherwise the machine will have peaked to the dataset and now we are `teaching to the test`

Dateset sizes

How to chose the sizes for our dataset? there is no one-fit-all solution but a good start is 50-25-25
We'll soon see evaluation tools called `learning curves` . These give us an indication of what happens to our validation-testing performance as we train on more and more examples. we will see a plateau in the performance. If that plateau happens within the 50% split size things are looking pretty good for us. However, imagine a scenario where we need 90% of our available data to get a decent performance.Then the 50-25-25 is not a good option

Parameters and Hyperparameters

Choosing between different machines (3-NN or 10-NN) in the same overall class of machine (k-NN) is selecting a hyperparameter. Selecting hyperparameters, like selecting models, is done in the selection phase.
Keep this distinction clear: `parameters` are set as part of the learning method in the training phase while `hyperparameters` are beyond the control of the learning method.

If we imagined the phases as loops (outer to inner)--> Assessment, Selection, Training. Then, adjusting hyperparameters means stepping out one level from adjusting the parameters—stepping out from Training to Selection.

Table 5.1 Phases and datasets for learning.

Phase	Name	Dataset Used	Machine	Purpose
inner	training	training set	set knobs	optimize parameters
middle	selection	validation test set	choose machines	select model, hyperparameters
outer	assessment	hold-out test set	evaluate performance	assess future performance

Overfitting and Underfitting

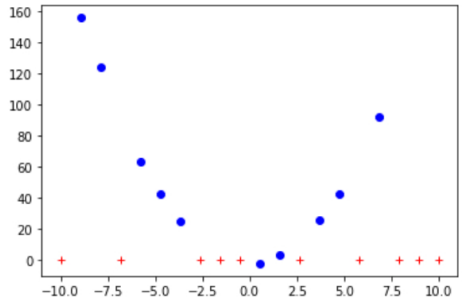
Synthetic Data and Linear Regression

Here, we'll make a trivial dataset with one feature and a target, and make a train-test split on it. Our noise is chosen uniform from values between −2 and 2.

In [149]:
```
N = 20
ftr = np.linspace(-10, 10, num=N) # ftr values
tgt = 2*ftr**2 - 3 + np.random.uniform(-2, 2, N) # tgt = func(ftr)

(train_ftr, test_ftr, train_tgt, test_tgt) = skms.train_test_split(ftr, tgt, test_size=N//2) # 50-50

display(pd.DataFrame({"ftr":train_ftr, "tgt":train_tgt}).T)
```

	0	1	2	3	4	5	6	7	8	9
ftr	6.8421	1.5789	-4.7368	-3.6842	-7.8947	0.5263	-8.9474	3.6842	-5.7895	4.7368
tgt	92.1354	3.0754	42.5621	24.6193	123.4699	-2.3648	155.6274	25.5555	62.9506	42.0830

In [150]:
```
plt.plot(train_ftr, train_tgt, 'bo')
plt.plot(test_ftr, np.zeros_like(test_ftr), 'r+');
```



In [151]:
```
# note: sklearn *really* wants 2D inputs (a table)
# so we use rehape here.

sk_model = linear_model.LinearRegression()
sk_model.fit(train_ftr.reshape(-1, 1), train_tgt)
```

```
sk_preds = sk_model.predict(test_ftr.reshape(-1, 1))
sk_preds[:3]
```

Out[151]:
```
array([97.7441, 13.2153, 82.8273])
```

We're not evaluating these predictions in any way. But at least—like our training targets—they are positive values.

Manually Manipulating Model Complexity

There are many other packages for finding those ideal knob values.
One of those alternatives is the `polyfit` routine in NumPy. It takes input and output values, our features and a target, and a degree of polynomial to align with the data.
It figures out the right knob values—actually, the coefficients of polynomials we then `np.poly1d` turns those coefficients into a function that can take inputs and produce outputs. Let's explore how it works:

In [152]:
```
# fit-predict-evaluate a 1D polynomial (a line)
model_one = np.poly1d(np.polyfit(train_ftr, train_tgt, 1))
preds_one = model_one(test_ftr)
print(preds_one[:3])
```

```
[97.7441 13.2153 82.8273]
```

Interesting. The first three predictions are the same as our LR model. Are all of the predictions from these inputs the same? Yes. Let's demonstrate that and calculate the RMSE of the model:

In [153]:
```
# the predictions come back the same
print("all close?", np.allclose(sk_preds, preds_one))

# and we can still use sklearn to evaluate it
mse = metrics.mean_squared_error
print("RMSE:", np.sqrt(mse(test_tgt, preds_one)))
```

```
all close? True
RMSE: 95.7052892630647
```

Great. So, two take-home messages here.

- we can use alternative systems, not just `sklearn`, to learn models. We can even use those alternative systems with `sklearn` to do the evaluation.
- `np.polyfit`, as its name implies, can easily be manipulated to produce any degree of polynomial we are interested in.

In [154]:
```
# fit-predict-evaluate a 2D polynomial (a parabola)
model_two = np.poly1d(np.polyfit(train_ftr, train_tgt, 2))
preds_two = model_two(test_ftr)
print("RMSE:", np.sqrt(mse(test_tgt, preds_two)))
```

```
RMSE: 1.6731511211804742
```

Hey, What an improvement :)

In [155]:
```
# Let's try a higher order
model_three = np.poly1d(np.polyfit(train_ftr, train_tgt, 9))
preds_three = model_three(test_ftr)
print("RMSE:", np.sqrt(mse(test_tgt, preds_three)))
```

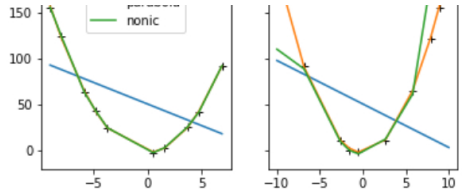
```
RMSE: 408.46418113650367
```

Never mind

Visualizing Overfitting, Underfitting, and "Just Right"

That didn't exactly go as planned. We didn't just get worse. We got utterly, terribly, horribly worse. What went wrong? We can break down what happened in the training and testing data visually:

In [156]:
```
fig, axes = plt.subplots(1, 2, figsize=(6, 3), sharey=True)

labels = ['line', 'parabola', 'nonic']
models = [model_one, model_two, model_three]
train = (train_ftr, train_tgt)
test = (test_ftr, test_tgt)

for ax, (ftr, tgt) in zip(axes, [train, test]):
    ax.plot(ftr, tgt, 'k+')
    for m, lbl in zip(models, labels):
        ftr = sorted(ftr)
        ax.plot(ftr, m(ftr), '-', label=lbl)

axes[1].set_ylim(-20, 200)
axes[0].set_title("Train")
axes[1].set_title("Test");
axes[0].legend(loc='upper center');
```



`model_one`, the straight line, has great difficulty because our real model follows a curved trajectory. `model_two` eats that up: it follows the curve just about perfectly. `model_three` seems to do wonderfully when we train. It basically overlaps with both model_two and the real outputs. However, it has problems when we go to testing. Since it is easy to add another midway model, I'll also throw in a degree-6 model.

In [157]:
```
results = []
for complexity in [1, 2, 6, 9]:
    model = np.poly1d(np.polyfit(train_ftr, train_tgt, complexity))
    train_error = np.sqrt(mse(train_tgt, model(train_ftr)))
    test_error = np.sqrt(mse(test_tgt, model(test_ftr)))
    results.append((complexity, train_error, test_error))

columns = ["Complexity", "Train Error", "Test Error"]
results_df = pd.DataFrame.from_records(results,
columns=columns,
index="Complexity")

results_df
```

Out[157]:

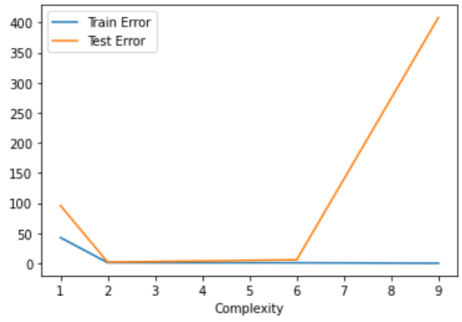
Complexity	Train Error	Test Error
1	42.5386	95.7053
2	0.9180	1.6732
6	0.7500	5.6818
9	0.0000	408.4642

Let's review what happened with each of the three models with complexity 1, 2, and 9.

- **Model one** (Complexity 1—a straight line). Model one was completely outclassed. It was doomed from the beginning. The It is too biased towards flatness. The model is `underfitting`.
- **Model three** (Complexity 9—a wiggly 9-degree polynomial). Model three certainly had enough horsepower. We see that it does very well on the training data. In fact, it gets to the point where it is perfect on the training data. But it completely falls apart when it comes to testing. Why? Because it memorized the noise —the randomness in the data. It varies too much with the data. We call this `overfitting`.
- **Model two** (Complexity 2—a parabola). Here we have the Goldilocks solution: it's not too hot, it's not too cold, it's `just right`.We do well enough on the training data and we see that we are at the lowest testing error. If we had set up a full validation step to select between the three machines with different complexity, we would be quite happy with model two. Model two doesn't exactly capture the training patterns because the training patterns include noise.

Let's graph out the results on the train and test sets:

In [158]:
```
results_df.plot();
```



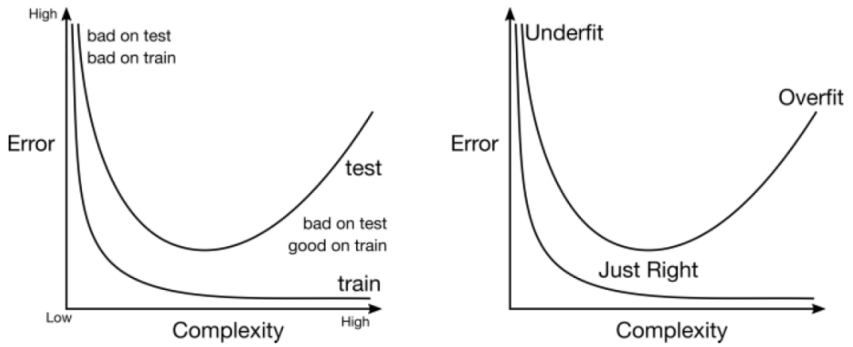


Figure 5.4 As complexity increases, we generally move from underfitting to just right to overfitting.

Here are the key points:

- **Underfitting**: A very simple model may not be able to learn the pattern in the training data It also does poorly on the testing data.
- **Overfitting**: A very complex model may learn the training data perfectly. However, it does poorly on the testing data because it also learned irrelevant relationships in the training data.
- **Just-right**: A medium-complexity model performs well on the training and testing data.

From Errors to Costs

In our discussion of overfitting and underfitting, we compared model complexity and error rates, These two aspects, error and complexity, are intimately tied together

Loss

We'll construct a `loss function` that quantifies what happens when our model is wrong on a single example.
Then we build a funcioin to `training loss` function that measures how well our model does on the entire training set.

In [159]:
```
def training_loss(loss, model, training_data):
    ' total training_loss on train_data with model under loss'
    return sum(loss(model.predict(x.reshape(1, -1)), y) for x, y in training_data)

def squared_error(prediction, actual):
    ' squared error on a single example '
    return (prediction - actual)**2

# could be used like:
# my_training_loss = training_loss(squared_error, model, training_data)
```

In [160]:
```
knn = neighbors.KNeighborsRegressor(n_neighbors=3)
fit = knn.fit(diabetes.data, diabetes.target)

training_data = zip(diabetes.data, diabetes.target)

my_training_loss = training_loss(squared_error, knn, training_data)
print(my_training_loss)
```

```
[863792.3333]
```

In [161]:
```
mse = metrics.mean_squared_error(diabetes.target,knn.predict(diabetes.data))
print(mse*len(diabetes.data))
```

```
863792.3333333333
```

Cost

As we saw with overfitting, if we make our model more and more complex, we can capture any pattern—even pattern that is really noise, So, we need something that works against complexity and rewards simplicity. We do that by adding a value to the training loss to create a total notion of cost. Conceptually, cost = loss + complexity
The term we add to deal with complexity has several technical names:regularization, smoothing, penalization, or shrinkage. We'll just call it complexity.

We don't have to have a fixed idea of how to trade off error and complexity, We can consider it as another hyperparameter λ, Lambda represents that tradeoff.

In [162]:
```
def complexity(model):
    return model.complexity

def cost(model, training_data, loss, _lambda):
    return training_loss(m,D) + _lambda * complexity(m)
```

Our cost goes up if:

- if we make more mistakes
- if we have a higher complexity

If we take λ = 2, one unit of complexity is comparable to two units of loss. If we take λ = .5, two units of complexity are comparable to one unit of loss. Shifting λ adjusts how much we care about errors and complexity.

Score

we can consider losses and scores to be inverses: as one goes up, the other goes down.So, we generally want a `high score` or a `low loss`.

(Re)Sampling: Making More from Less

If we ask people to estimate the number of beans in a jar, they will individually be wrong. But if we get many, many estimates, we can get a better overall answero, how do we generate multiple estimates for evaluation? We need multiple datasets. But we only have one dataset available. How can we turn one dataset into many?

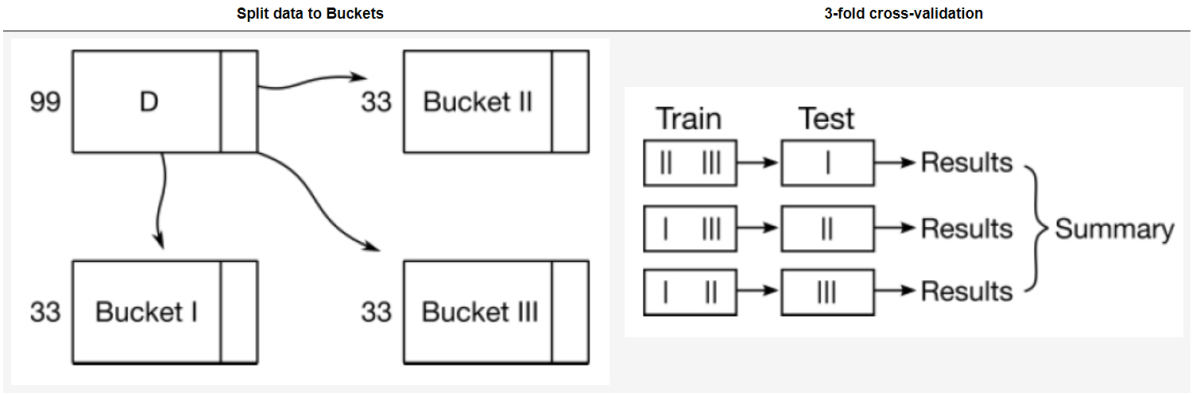
Cross-Validation

Cross-validation is like a card game where we deal out all the cards to three players, play a round of the game, and then shift our cards to the player on the right—and repeat that process until we've played the game with each of the three different sets of cards.

Let's go directly to an example. Cross-validation takes a number of `folds` which is like the *number of players* we had above. With three players, or three folds, we get three different attempts to play the game. For 3-fold cross-validation
we'll take an entire set of labeled data and shake it up. We'll let the data fall randomly—as evenly as possible—into the three buckets in Figure 5.5 labeled with Roman numerals: B_I, B_{II}, and B_{III}.

Now, we perform the following steps:

1. Take bucket $\mathbf{B_I}$ and put it to the side. Put $\mathbf{B_{II}}$, **and** $\mathbf{B_{III}}$ together and use them as our training set. Train a model `ModelOne` from that combined training set. Now, evaluate `ModelOne` on bucket BI and record the performance as `EvalOne`.
2. Take $\mathbf{B_{II}}$ and put it to the side. Put buckets $\mathbf{B_I}$, **and** $\mathbf{B_{III}}$ together and use them as our training set. Train `ModelTwo` from that combined training set. Now, evaluate ModelTwo on bucket $\mathbf{B_{II}}$ and record the performance as `EvalTwo`.
3. Take $\mathbf{B_{III}}$ and put it to the side. Put buckets $\mathbf{B_I}$, **and** $\mathbf{B_{II}}$ together and use them as our training set. Train `ModelThree` from that combined training set. Now, evaluate ModelTwo on bucket $\mathbf{B_{III}}$ and record the performance as `EvalTthree`.



In [2]:
```
# Here is an example for 5-CV

# data, model, fit & cv-score
model = neighbors.KNeighborsRegressor(10)
skms.cross_val_score(model, diabetes.data, diabetes.target, cv=5, scoring='neg_mean_squared_error')

# notes:
# defaults for cross_val_score are
# cv=5 fold, no shuffle, stratified if classifier
# model.score by default (regressors: R^2, classifiers: accuracy)
```

Out[2]:
```
array([-3206.7542, -3426.4313, -3587.9422, -3039.4944, -3282.6016])
```

what's up with the `scoring='neg_mean_squared_error'` argument? Remember when we talked about the score, we want to have a large score and a small error.So sklearn negate the error measure to a score measure

all the scores here are negative but a bigger score is better.Think of it like money you want to be down by $5 instead of $50

Note that if the estimator is a classifier and y is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used.

Stratification

In [2]:
```
iris = datasets.load_iris()
model = neighbors.KNeighborsClassifier(10)
skms.cross_val_score(model, iris.data, iris.target, cv=5)
```

Out[2]:
```
array([0.9667, 1.    , 1.    , 0.9333, 1.    ])
```

the cross-validation was done in a stratified manner because it is the default for classifiers in sklearn. What does that mean? Basically, stratification means that when we make our training-testing splits for cross-validation, we want to respect the proportions of the targets that are present in our data. Huh? Let's do an example.

In [6]:
```
pet = np.array(['cat', 'dog', 'cat', 'dog', 'dog', 'dog'])

list_folds = list(skms.KFold(2).split(pet))
training_idxs = np.array(list_folds)[:, 0, :]

print(pet[training_idxs])
```

```
[['dog' 'dog' 'dog']
 ['cat' 'dog' 'cat']]
```

Note that there is no cat in the first fold

In [8]:
```
# stratified
# note: typically this is behind the scenes
# making StratifiedKFold produce readable output
# requires some trickery. feel free to ignore.
pet = np.array(['cat', 'dog', 'cat', 'dog', 'dog', 'dog'])
idxs = np.array(list(skms.StratifiedKFold(2).split(np.zeros_like(pet), pet)))
```

```
idxs = np.array(list(skms.StratifiedKFold(2).split(np.ones_like(pet), pet)))
training_idxs = idxs[:, 0, :]
print(pet[training_idxs])
```

```
[['cat' 'dog' 'dog']
 ['cat' 'dog' 'dog']]
```

Note that there is a balanced number of cats and dogs in both folds qual to their proportion in the overall dataset

Stratification is particularly useful when

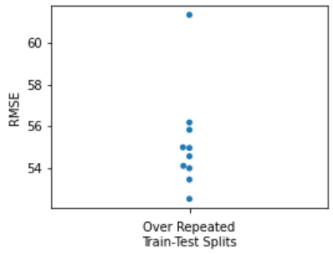
1. we have limited data overall or
2. we have classes that are poorly repres ented in our dataset

Repeated Train-Test Splits

In [11]:
```
# as a reminder, these are some of the imports
# that are hidden behind: from mlwpy import *
# from sklearn import (datasets, neighbors,
# model_selection as skms,
# linear_model, metrics)
# see Appendix A for details

linreg = linear_model.LinearRegression()
diabetes = datasets.load_diabetes()

scores = []
for r in range(10):
    tts = skms.train_test_split(diabetes.data, diabetes.target, test_size =.25)
    (diabetes_train_ftrs, diabetes_test_ftrs, diabetes_train_tgt, diabetes_test_tgt) = tts

    fit = linreg.fit(diabetes_train_ftrs, diabetes_train_tgt)
    preds = fit.predict(diabetes_test_ftrs)

    score = metrics.mean_squared_error(diabetes_test_tgt, preds)
    scores.append(score)

scores = pd.Series(np.sqrt(sorted(scores)))
df = pd.DataFrame({'RMSE':scores})
df.index.name = 'Repeat'
display(df.T)
```

Repeat	0	1	2	3	4	5	6	7	8	9
RMSE	52.5422	53.4667	54.0062	54.1177	54.5769	54.9739	55.0024	55.8374	56.1877	61.3167

In [12]:
```
ax = plt.figure(figsize=(4, 3)).gca()
sns.swarmplot(y='RMSE', data=df, ax=ax)
ax.set_xlabel('Over Repeated\nTrain-Test Splits');
```



In [13]:
```
display(df.describe().T)
```

	count	mean	std	min	25%	50%	75%	max
RMSE	10.0000	55.2028	2.4023	52.5422	54.0341	54.7754	55.6287	61.3167

In [14]:
```
np.mean(df['RMSE'])
```

Out[14]:
```
55.20277416011961
```

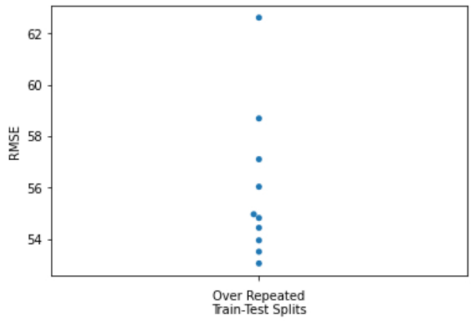
A Better Way and Shuffling

Fortunately, sklearn has done that. If we pass in a `ShuffleSplit` data-splitter to the cv argument of cross_val_score, we get precisely the algorithm we hand-coded above.

In [16]:
```
linreg = linear_model.LinearRegression()
diabetes = datasets.load_diabetes()

# nondefault cv= argument
ss = skms.ShuffleSplit(test_size=.25) # default, 10 splits
scores = skms.cross_val_score(linreg, diabetes.data, diabetes.target, cv=ss, scoring='neg_mean_squared_error')

scores = pd.Series(np.sqrt(-scores))
df = pd.DataFrame({'RMSE':scores})
df.index.name = 'Repeat'
display(df.describe().T)
```

```
ax = sns.swarmplot(y='RMSE', data=df)
ax.set_xlabel('Over Repeated\nTrain-Test Splits');
```

	count	mean	std	min	25%	50%	75%	max
RMSE	10.0000	55.9133	2.9063	53.0497	54.0687	54.8888	56.8316	62.6032

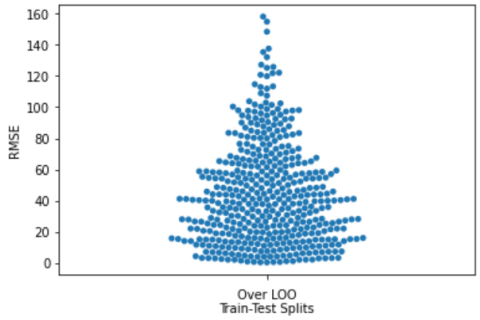


Leave-One-Out Cross-Validation

we could take an extreme approach to cross-validation and use as many cross-validation buckets as we have examples.
with 20 examples, we could potentially make 20 train-test splits, do 20 training fits, do 20 testing rounds, and get 20 result. This version of CV is called leave-one-out cross-validation (LOOCV)

In [18]:
```
linreg = linear_model.LinearRegression()
diabetes = datasets.load_diabetes()

loo = skms.LeaveOneOut()
scores = skms.cross_val_score(linreg, diabetes.data, diabetes.target, cv=loo, scoring='neg_mean_squared_error')

scores = pd.Series(np.sqrt(-scores))
df = pd.DataFrame({'RMSE':scores})
df.index.name = 'Repeat'

display(df.describe().T)

ax = sns.swarmplot(y='RMSE', data=df)
ax.set_xlabel('Over LOO\nTrain-Test Splits');
```

	count	mean	std	min	25%	50%	75%	max
RMSE	442.0000	44.3557	32.1973	0.2075	18.4820	39.5472	63.9738	158.2355



Break-It-Down: Deconstructing Error into Bias and Variance

Let's take an example of a race track where we measure the time the cars take for two laps (d = 2) , So the information that we have is (d, t_1, t_2). How can we relate these values together? we could get $\mathbf{t_{tot} = t_1 + t_2}$, and get the average speed for each car/driver (speed = distance / total time)

Driver	t_1	t_2	t_{total}	s	d
Mario	35	75	110	.018	2
Luigi	20	40	60	.033	2
Yoshi	40	50	90	.022	2

Well is our measurement perfect or there could be errors? Let's try different possibilities

Inputs	**Output**	**Measurement errors**	**True Relationship**	**Try to Relate With**	**Perfect?**	**Why?**
t_1, t_2	t_{total}	no	add	add	yes	
t_1, t_2	t_{total}	yes	add	add	no	measure-ment

						errors
t_{total}, s	d	no	multiply	add	no	can't get right form

So what we learnt that we might not get a good estimate (perfect = no) in two cases:

1. if there is a error in the input data *(we should deal with before feeding to the training model)*
2. the interaction between the training data and the machine learning model *(here the model relate with addition instead of multiplication)*

Variance of the Data

when we have an incorrect class or a MSE greater than zero here can be a few different causes.

1. the actual randomness in the relationship between the input features and the output target which is somthing that we don't have control over *not every of clerk with 5 years experience have the same income*, we could include more features to narrow the gab but there will always be an error
2. Having a range of outputs, a single input might relate to more than one output *such as throwing a dice*

Variance of the Model

Is the way the models vary due to the random selection of the data we train on. There is some errors that we can control, let's say we have a 1-NN model and one outlier point, then the model will estimate wrongle but if we have a higher k-NN model the error won't be as powerful

Bias of the Model

Our last source of error is where we have the most control. When I choose between two models, one may have a fundamentally better resonance with the relationship between the inputs and outputs, Like choosing a 2-order polynomial instead of a line
We say that a model that cannot match the actual relationship between the inputs and outputs higher bias, ***A highly biased model has difficulty capturing complicated patterns*** A model with low bias can follow more complicated patterns.

All Together Now

Error = Bias$_{Learner}$ + Variance$_{Learner(Training)}$ + Variance$_{Data}$

Examples of Bias-Variance Tradeoffs

Bias-Variance for k-NN

If we have only 10 example in our set

Let's start with 1-NN model If we have a new example we find the closest value to it and will assign it with the same target, we can see where could this become a problem. Every training example gets to have its own say without consulting anyone else. So looking at the 10 points we have once we find the colsest the other 9 are useless

Let's see a 10-NN model If we have a new example we find the closest 10, average them to find the target label, But the problem is that we only started with 10 so basically we are getting the average for the whole dataset and that target label won't change for any new point

So Increasing the number of neighbors increases our bias and decreases our variance. Decreasing the number of neighbors increases our variance and decreases our bias.

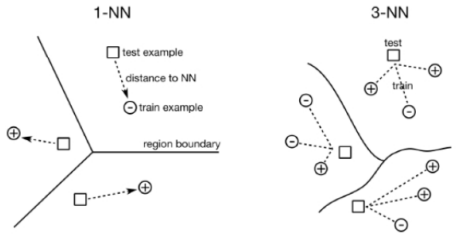


Figure 5.9 Bias in k-NN.

Bias-Variance for Linear Regression

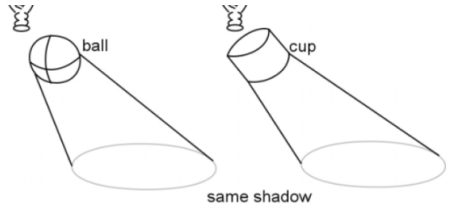
we will compare between two models

1. Flat horizotal plane -- *ConstantLinear*
2. Plan that could have a tilt -- *PlainLinear*

So we could have 4 possible wight values

- Constant linear: include no features, $w_i = 0$ for all i =6 0.
- Few: include a few features, most $w_i = 0$.
- Many: include many features, a few $w_i = 0$.
- Plain linear: include all features, no $w_i = 0$.

on one end When we set some or all $w_i = 0$ we lose the information given by this feature which increase our bias



gure 5.10 Losing features (dimensions) restricts our view of the world and increases our bias.

on the other hand if we extended our features by adding adding features—like polynomial terms (x^2, $x_1 * x_1$) — will alow us to capure examples that appear to be oddities but also we will be fooled by noise

In linear regression, adding features—like polynomial terms—decreases bias but increases variance. Conversely, forcing the weights of features to zero increases bias and decreases variances.

> Consulting more examples in nearest neighbors leads to more bias, yet consulting more features in linear regression leads to less bias.

Summary

Scenario	Example	Good	Bad	Risk
high bias & low variance	more neighbors	resists noise	misses pattern	underfit
	low-degree polynomial	forced to generalize		
	smaller or zero linear regression coefficients			
	more independence assumptions			
low bias & high variance	fewer neighbors	follows complex patterns	follows noise	overfit
	high-degree polynomial		memorizes training data	
	bigger linear regression coefficients			
	fewer independence assumptions			