Model Training and Improvement

How to train your model

Yordan Darakchiev

Technical Trainer

iordan93@gmail.com

Table of Contents

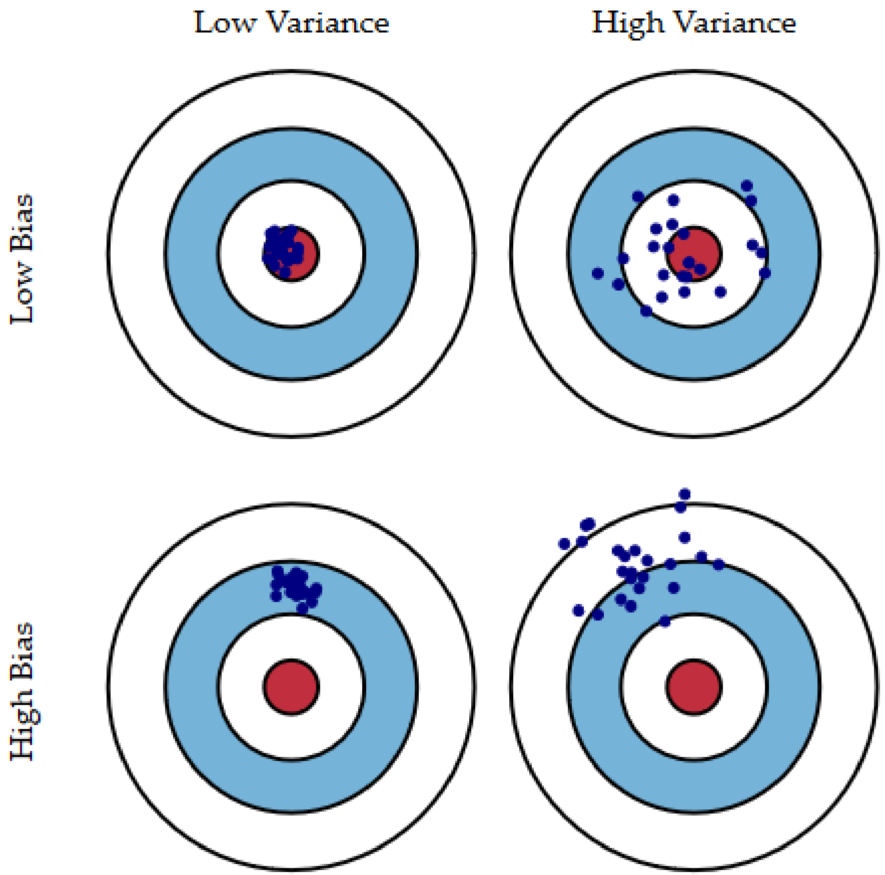
- sli.do: #model-impr
- Bias – variance tradeoff
    - Applying regularization
- Training and testing
- Cross-validation
- Model tuning and selection
- Feature selection, feature engineering

Regularization

Taming your model

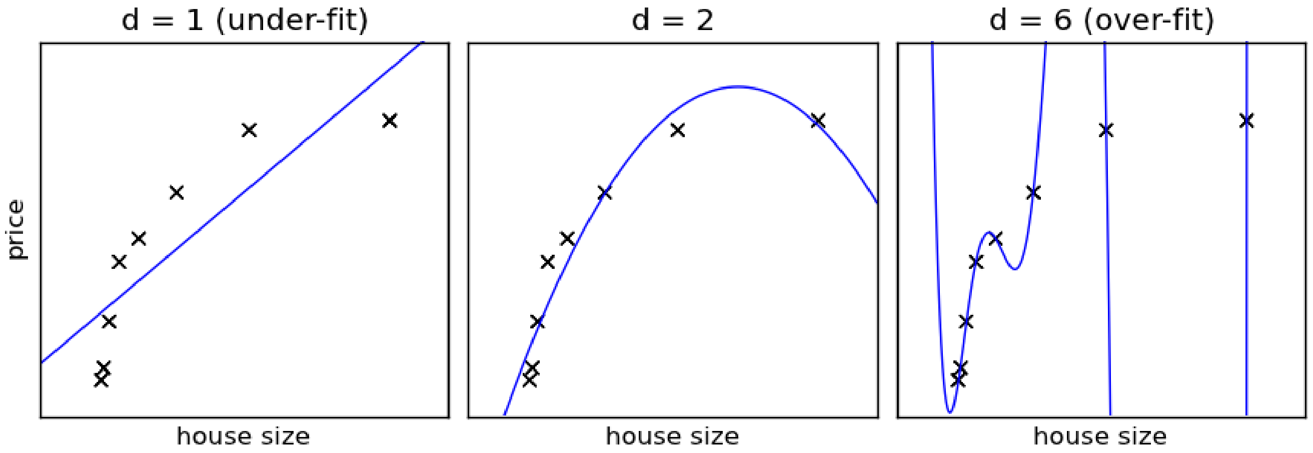
Bias-Variance Tradeoff

- When we fit models, we have two main sources of errors
  - Bias – how far are the predicted from the actual values
  - Variance – variability of prediction for a certain data point
- Illustration – shooter aimed at a bullseye target
  - High bias – the aim is shifted away from the center
  - High variance – the points are more "spread out"



Bias-Variance Tradeoff (2)

- When we fit several models, they perform differently
  - Some are not complex enough (don't describe data well enough)
    - Underfitting (high bias)
  - Some may describe the data "too well" and **fail to generalize** when **new** data points are introduced
    - Overfitting (high variance)
- Optimal model: tradeoff between underfitting and overfitting
  - Usually underfitting is easy to spot (poor performance w.r.t. some metric)
  - Overfitting is more complicated
    - Many methods exist to prevent overfitting



Regularization

- Method for finding a good bias-variance tradeoff
  - Filter out noise from data
  - Handle highly correlated features
- **L2** regularization – "second norm" (Euclidean): $\lambda||w||_2^2 \equiv \lambda\sum_{j=1}^{m} w_j^2$
  - λ – regularization parameter
  - Shrinks all model weights by the same value
- **L1** regularization – "first norm": $\lambda||w||_1 \equiv \lambda\sum_{j=1}^{m} |w_j|$
  - Sets some coefficients to 0: feature selection
- In the ideal case, we can use both L1 and L2
- Usage: add the regularization term to the cost function
  - $\lambda > 0$, larger ⇒ stronger regularization

Linear Regression with Regularization

- Ridge regression – L2
  - Cost function: $J(w) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2 + \lambda||w||_2^2$

```python
from sklearn.linear_model import Ridge
model = Ridge(alpha = 0.01)
```

- LASSO (**L**east **A**bsolute **S**hrinkage and **S**election **O**perator) – L1
  - Cost function: $J(w) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2 + \lambda||w||_1$

```python
from sklearn.linear_model import Lasso
model = Lasso(alpha = 0.01)
```

- Elastic Net
  - Has both regularization terms

```python
from sklearn.linear_model import ElasticNet
model = ElasticNet(alpha = 1.0, l1_ratio = 0.5)
```

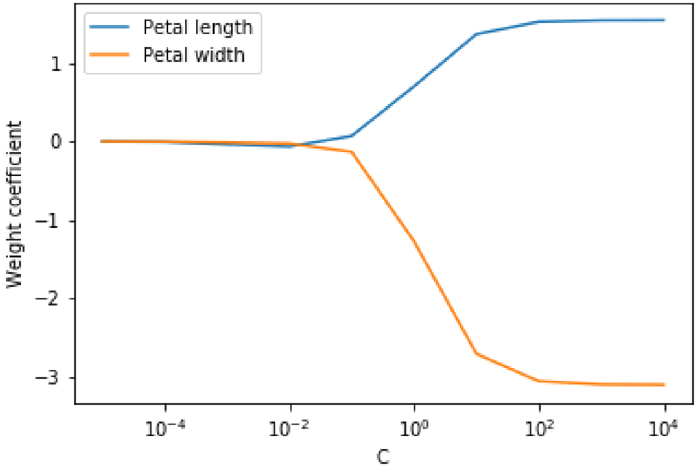
Logistic Regression with Regularization

- The `LogisticRegression` class in `scikit-learn` accepts a regularization parameter C
  - Added to the cost function, L2-type
  - $C = \frac{1}{\lambda}$ (by convention)
- Increasing C ⇒ weaker regularization
  - The algorithm follows the data more closely
- Decreasing C ⇒ stronger regularization
  - The algorithm "doesn't care too much about the data"
- How to choose a value?
  - We'll talk about this later, stay **tuned**...

Visualizing Regularization Parameters

- Perform a logistic regression on the Iris dataset with several values of C
  - Display the weights
    - Observe how decreased C leads to weight shrinking

```
iris = load_iris()
attributes, labels = iris.data[:, [2, 3]], iris.target
weights, params = [], []
for c in np.arange(-5, 5):
model = LogisticRegression(C = 10.0 ** c)
model.fit(attributes, labels)
weights.append(model.coef_[1]) # Display only the second class
params.append(10.0 ** c)
weights = np.array(weights)
plt.plot(params, weights[:, 0], label = "Petal length")
plt.plot(params, weights[:, 1], label = "Petal width")
plt.xlabel("C")
plt.ylabel("Weight coefficient")
plt.xscale("log")
plt.legend()
plt.show()
```



Model Testing

Seeing how well your model performs on new data

Training and Testing Sets

- One of the most important rules in machine learning is
  - **NEVER test the model with the data you trained it on!**
    - The model may "cheat" and learn the answers instead of finding structure in the data
- Since we usually have one dataset, it's useful to "hold out" some of the data for testing
  - Usually 70% of the data is for training and 30% – for testing
  - We need to take randomized samples
  - In cases of classification, we need stratified samples
- scikit-learn has a convenient method for this

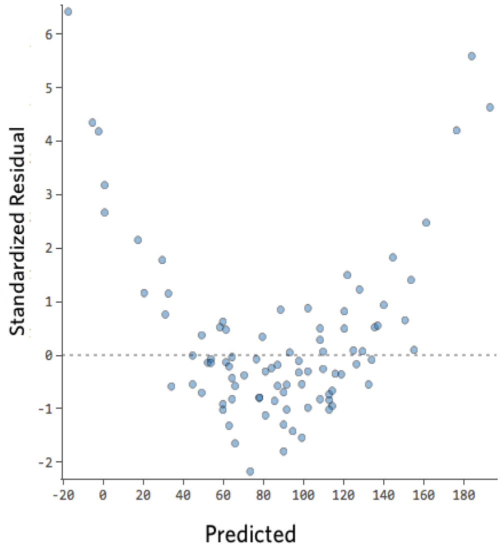
```
from sklearn.model_selection import train_test_split
attr_train, attr_test, l_train, l_test = train_test_split(attributes, labels, train_size = 0.7)
```

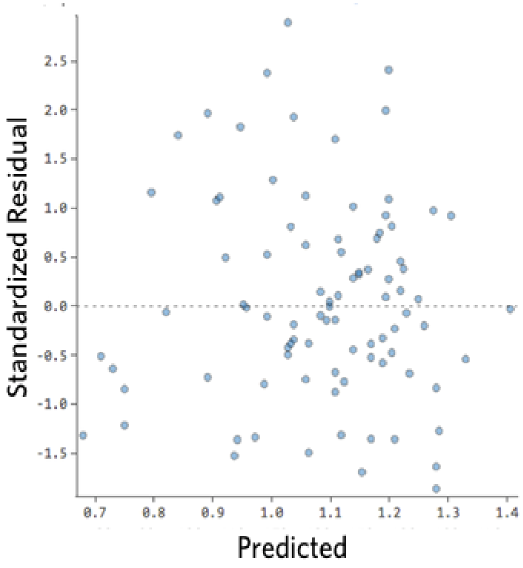
Evaluating Model Performance

- Once we train the model, we use the test data to score it
  - Using one of the scoring metrics
- Scoring metrics
  - **Regression:** usually coefficient of determination R^2 – proportion of variance predictable from the independent variables
    - Other: mean squared error, mean absolute error, explained variance
  - **Classification:** usually accuracy (how many items have been properly classified)
    - Other: precision, recall, F1
- For more metrics, look at the docs
- The output from scoring tells us how good the model is

Evaluating Regression

- No fixed rules, use your intuition and knowledge about the data
- Several guidelines
  - It's good to display more than one metric
    - For example, mean squared error and coefficient of determination
    - Also useful: mean absolute error and mean squared error
  - Create a residual plot ($O - E$: observed minus estimated)
    - There should be no visible structure
    - If there is some structure in the residuals, the model fails to explain something
  - Create a histogram of the residuals
    - Most residuals should be "sufficiently close" to zero





Evaluating Classification

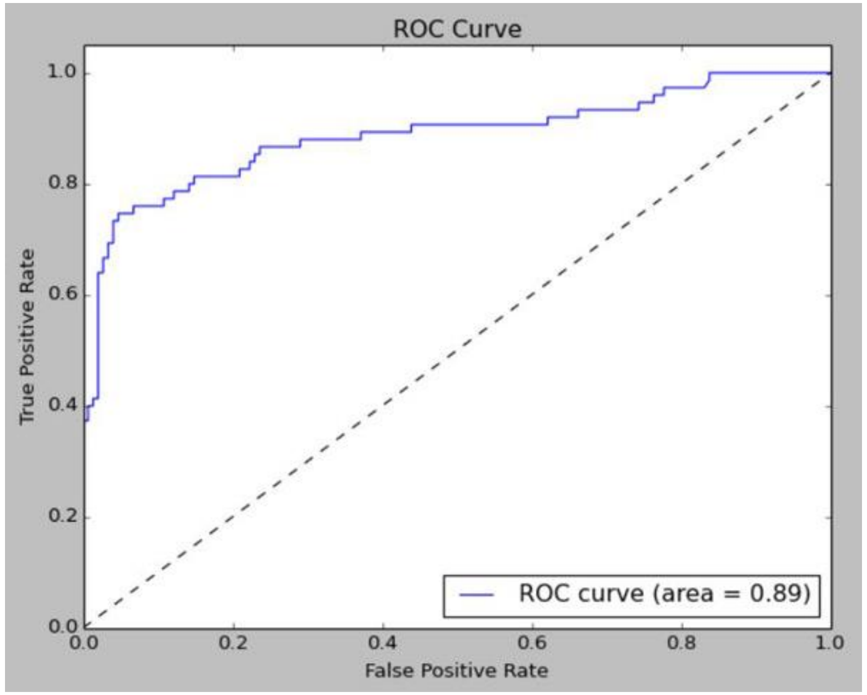
- Confusion matrix (error matrix)
  - Shows predicted vs. actual classes
  - Simplest case: 2-class classifier
    - Can be extended
  - FP = Type I error, FN = Type II error

	p' (Predicted)	n' (Predicted)
p (Actual)	True Positive	False Negative
n (Actual)	False Positive	True Negative

- Metrics: numbers derived from the confusion matrix
  - Accuracy (number of correctly classified samples): $\mathrm{A} = \frac{TP+TN}{TP+TN+FP+FN}$
    - If detecting anomalies, accuracy can be misleading
  - Precision (how many selected samples are relevant): $\mathrm{P} = \frac{TP}{TP+FP}$
  - Recall (how many relevant samples are selected): $\mathrm{R} = \frac{TP}{TP+FN}$
  - F1-score: $\frac{2TP}{2TP+FP+FN} = 2\frac{R.P}{R+P}$
  - Many more metrics exist (useful for specific cases)

Receiver Operating Characteristic

- Limited to 2-class classification
  - We can use "1 vs. all" for more classes
- A plot of true positive rate vs. false positive rate
  - A "bisector line" represents truly random guessing
  - Any curve above the line is better than random
    - Closer to the upper left corner = better
    - Below the line: still better than random, we have to reverse the classifier output



- Area under the curve (AUC): closer to 1 = better

Cross-Validation

- Most algorithms improve their parameters based on the test scores
    - This means knowledge of test data may "leak" into the algorithm and overfit the data
- Solution: cross-validation
    - Split all data into k groups (folds) – usually $k = 10$
        - **More** samples = **fewer** folds
        - Using a `KFold` splitter
    - Each time test with $k - 1$ folds and test with the other fold

```
from sklearn.model_selection import StratifiedKFold, cross_val_score

k_fold = StratifiedKFold(n_splits = 5)
```

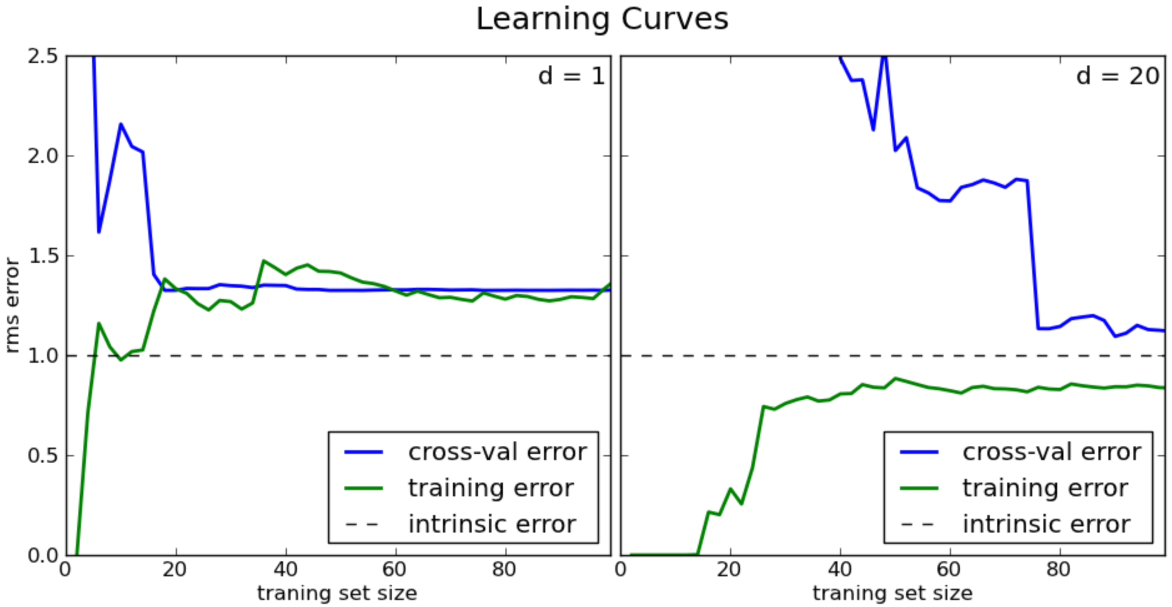
```
scores = cross_val_score(model, attributes, labels, cv = k_fold)
```

Improved Technique

- Test / train set → faster
- Cross-validation → more accurate
- Best performance (but even slower): combine the methods
  - Leave out some of the data for testing at the beginning (e.g. 30%)
  - Perform cross-validation on the other 70%
  - Fine-tune the model and / or select one of many models based on the best cross-validation score
  - Run the best model on the other 30%
    - We selected the best model based on the training data ⇒ we have some bias
      - One model will always have the highest score, even if it's by chance
    - This truly out-of-sample method removes (some of) the bias
  - **Model selection:** choose the best performing model

Learning and Validation Curves

- Plots which allow us to diagnose bias and variance problems
  - Some metric (e.g. accuracy) vs. a model parameter (e.g. sample size)
  - Plot two curves – for the training and validation data
- High bias – accuracy for training and validation is too low
  - Solution: add more model features, decrease regularization
- High variance – large gap between the two curves
  - Solution: remove model features (preprocessing / feature selection / feature engineering, etc.), increase regularization
- Tutorial (`scikit-learn`)



Hyperparameter Tuning

- Techniques for choosing the best model hyperparameters
  - Such as regularization
- Most widely used: grid search
  - "Brute-force": specify parameters; run models with all possible parameter combinations; choose the best model
- Randomized search – each setting is sampled randomly from a parameter range
  - Faster but not guaranteed to produce the best results

```
from sklearn.model_selection import GridSearchCV

tuned_params = [{"C": [0.001, 0.01, 0.1, 1, 10, 100, 1000],
  "penalty": ["l1", "l2"]}]
grid = GridSearchCV(LogisticRegression(), tuned_params)
grid.fit(attr_train, l_train)
print(grid.best_params_) # Estimator: grid.best_estimator_
```

Making the Best of Our Models

- Usually we don't know right away which algorithm will perform the best
- We select several algorithms (e.g. for classification)
  - Fine-tune their parameters using grid search (or some other technique)
  - Select the best combination of parameters
- After that, we compare the best algorithms from each type
  - Using the model selection procedure
    - Hold-out set + cross-validation set
  - Select the best performing model type on the cross-validation set
    - Test it on the "hold-out" set
- Improvements: perform test / train split on the hyperparameter tuning step; use different performance scores

Manipulating Features

Making things simpler

Guidelines for Manipulating Features

- Main ideas
  - Reduce the number of features (⇒ simpler model)
  - Keep relevant information only
- Feature selection
  - Removing irrelevant features
  - Regularization does a good job at this
  - Other methods: dimensionality reduction
    - We'll talk about this later in the course
- Feature engineering
  - "Manually" coming up with new, meaningful features
    - Requires a lot of work and domain knowledge
  - A sample process
    - Brainstorm features
    - Create some features based on your ideas
    - Check how the features work with the model
    - Improve the new features if needed
    - Repeat all previous steps as many times as needed

Summary

- Bias – variance tradeoff
  - Applying regularization
- Training and testing
- Cross-validation
- Model tuning and selection
- Feature selection, feature engineering

Questions?