BROWN
Computer Science

CS1951A: Data Science

Lecture 17: Cross validation, Regularization and Feature Selection

Lorenzo De Stefani

Spring 2022

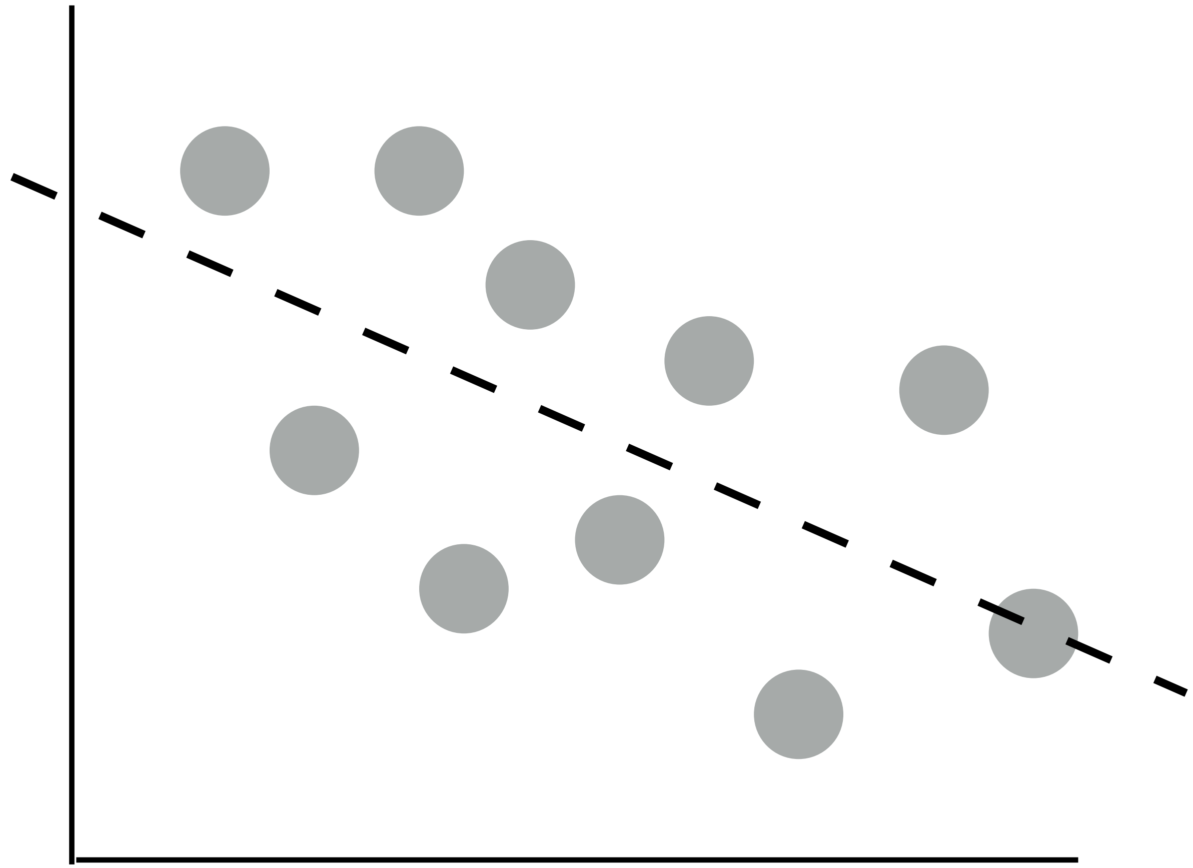
Outline

- Overfit
- Cross validation
- Regularization

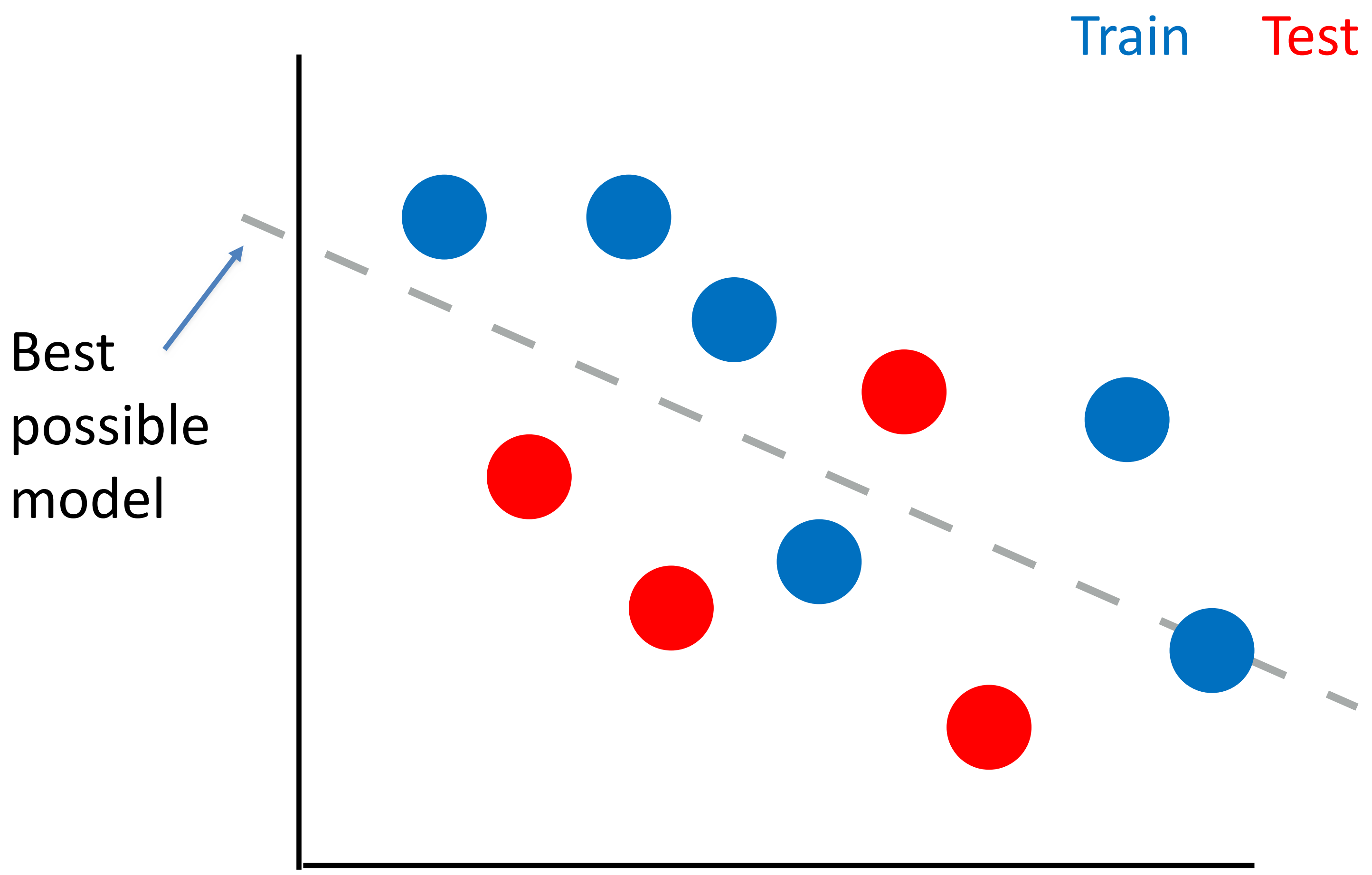
Train/Test Splits

- By definition, trained models are minimizing their objective for the data they see, but not for the data they don't see
- What we really care about is how the model generalizes to data we have not observed yet
- One common approach is to split our training data into disjoin sets—a train set and a test set—and assess performance on test given parameters set using train.

Train/Test Splits

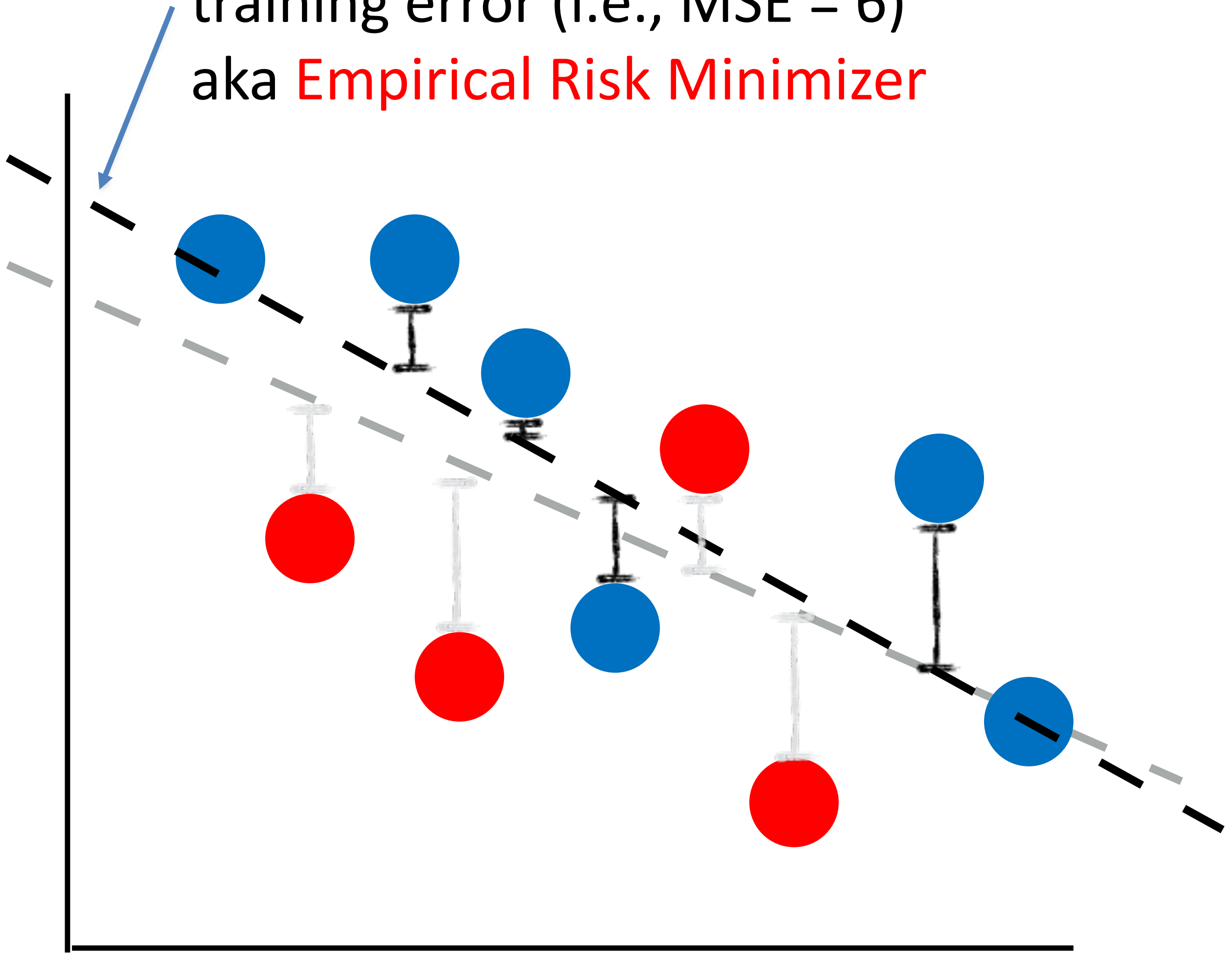


Train/Test Splits



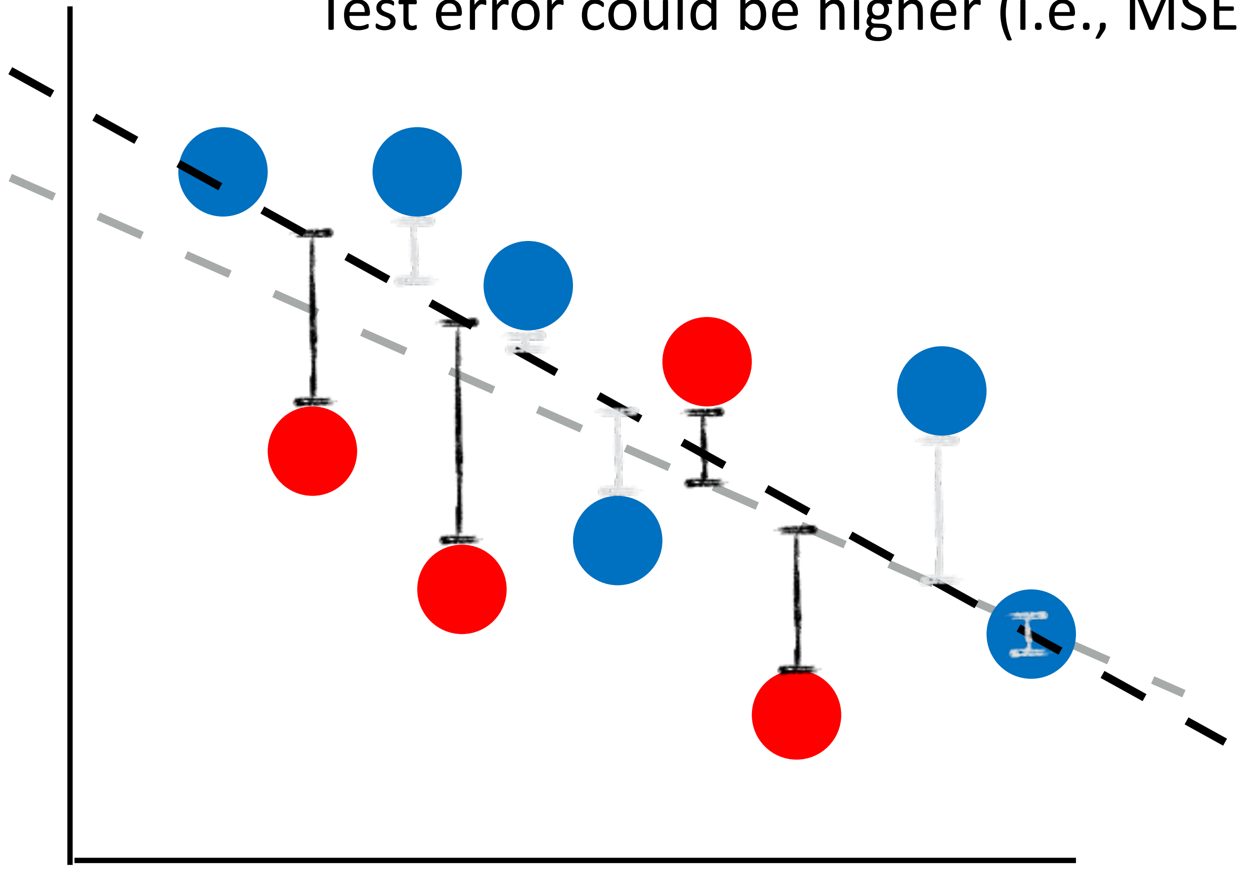
Train/Test Splits

Model that minimizes
training error (i.e., MSE = 6)
aka Empirical Risk Minimizer



Train/Test Splits

Test error could be higher (i.e., MSE = 6)



Generalization guarantees

- Generalization error:

$$E_G = |E_{train} - E_{test}|$$

- Statistical learning theory provides the tools to characterize the distribution of E_G

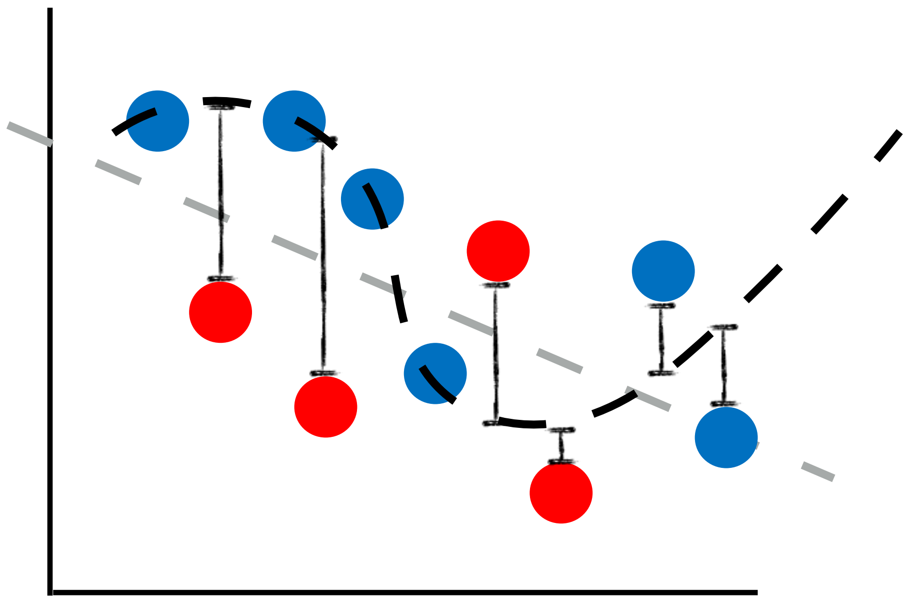
$$P(E_G > \epsilon) < \delta$$

- The distribution we can claim depends on parameters which capture the complexity of the class of models we are considering

Generalization guarantees

- These bounds are often rather pessimistic when compared with actual performance of ML algorithms
  - They state a much higher requirement for number of observations
  - They state much weaker guarantees than those observed
- Still, very important tool! We want to guarantee that something is going to work!

Complexity and overfitting



The more complex the possible models, the more likely we are to observe a large discrepancy between E_R and E_T

- The more complex the model, the more we tend to overfit to the training data
- In other words, we need more training samples to "learn well"

Cross Validation

- Some train/test splits are worse than others
  - Particularly unbalanced sets
- To get a more stable estimate of test performance, we can use cross validation
  1. Divide the data randomly in k distinct subsets of the same size (folds)
  2. In k-1 rounds select k-1 folds as the training set and the remaining one as the test set
  3. Compute the average generalization error

Cross Validation

- Some train/test splits are worse than others
  - Particularly unbalanced sets
- To get a more stable estimate of test performance, we can use cross validation

```
accs = []
for i in range(num_folds):
    train, test = random.split(data)
    clf.fit(train)
    accs.append(clf.score(test))
```

The complexity vs generalizability tradeoff

- The more complex the model, the more expressive
  - Captures more details about the model
- The more complex the model, the harder it is to "learn it"
  - The more examples we need to see
  - The more information we need to acquire
- While using complex models may seem appealing, we incur in the risk of overfitting to the data
  - We need to observe a high number of examples to have the same guarantees as if we had simpler models

Regularization

- Modify the cost function to add a cost for increasing the complexity of the model
  - E.g., In linear regression incur a cost for having more features (more non-zero weights), or for assuming features are very important (more high weights)
  - Or "early stopping"—for iterative training procedures (e.g., gradient descent) stop before the model has fully converged
    - We assume the final steps are spent memorizing noise

- By definition, regularization will make your model worse during training…
- But hopefully better at test time…
- Which is what you really care about!

Regularization

$$min_\theta\big(loss(x;\theta) + \lambda cost(\theta)\big)$$

- Adds an extra "hyperparameter" which controls how much you penalize for the complexity of the model

Norms

Given a vector $\vec{x} = (x_1, x_2, \dots)$

- l_0 norm $\quad l_0 = \sum_i x_i^0 \quad$ #non-zero coefficients
- l_1 norm: $\quad l_1 = \sum_i |x_i| \quad$ encourages sparsity
- l_2 norm: $\quad l_2 = \sqrt{\sum_i x_i^2} \quad$ more stable
- l_p norm: $\quad l_p = \sqrt[p]{\sum_i x_i^p}$

Regularization examples for linear regression

- Linear Regression — No regularization

$$min_w((\vec{y} - \vec{w} \cdot \vec{x})^2)$$

- Lasso Regression — Linear regression with l_1 penalty on the vector of coefficients

$$min_w((\vec{y} - \vec{w} \cdot \vec{x})^2 + \lambda l_1(\vec{w}))$$

- Ridge Regression— Linear regression with l_2 penalty on the loss

$$min_w((\vec{y} - \vec{w} \cdot \vec{x})^2 + \lambda l_2(\vec{w}))$$

- Logistic Regression usually uses l_1 or l_1 regularization by default (e.g. in sklearn)

Dev/Validation Sets

- Often you need to make meta-decisions, not just set the parameters
  - Which model is better (i.e. generalizes better to held out data)?
  - Which regularization to use?
  - How many training iterations?
- Do do this, you have to split data into training /developement/test
- If you use test data to set these hyper-parameters, you are "peaking" at unseen data in order to fit the model, and thus test performance is no longer actually representative of how you would do in the real world

Feature Selection

- Explicitly remove features from model before training
- Lots of heuristic techniques (no magic solutions, requires trial and error)
- Some techniques:
  - Remove correlated features
  - Remove low-variance features
  - Iteratively add features with highest weight or information gain
  - Iteratively remove features with lowest weight or information gain
  - Dimensionality Reduction (e.g. SVD, PCA)