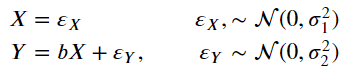
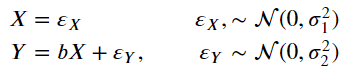
Prep Meeting 20

# Investigation of SEM

Last time we had a discussion. Assume a structural equation model with additive Gaussian noise. More concretely, assume



Now, from the data generation, we can clearly see that X affects Y, but not the way around. However, from just looking at data, how can we determine this? Long story short, we **cannot**. In short, the question is, is our true model the left model, or the right model?

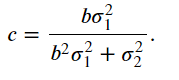
 .

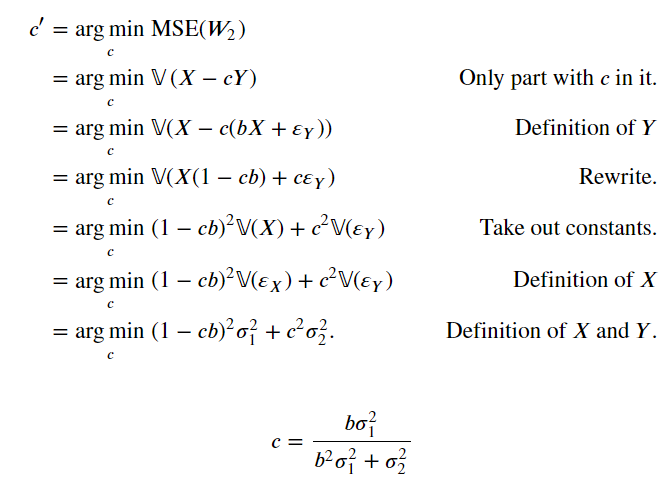
OR

1. **Value for c.** The first question, **what is c**? Intuitively, you would think that c would be 1 / b. So, if X = 2Y, then the c that minimizes the MSE on the training data would be c = 1 / b = 0.5. This is, however, not the case.

If σ2 is very large, then Y is almost purely random noise, no matter the values for b. Then, if we try to predict X using Y, we try to predict X using random noise. Obviously, the “optimal” choice would then be to pick c = 0, and not 1 / b.

Therefore, we see that the value for c depends on b and the ratio of noise, so the ratio of σ1 and σ2. It turns out that the expected value for c will be.

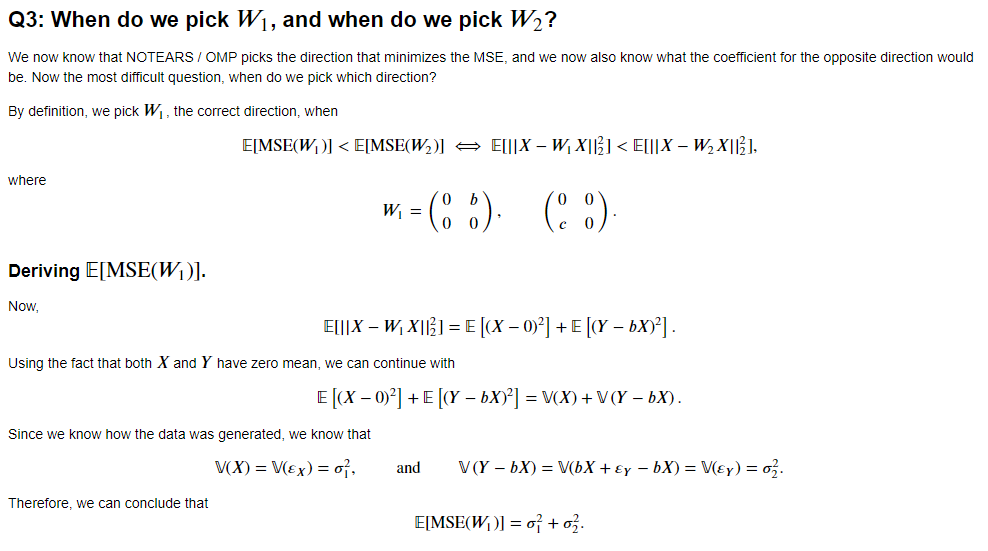
Derivation: .



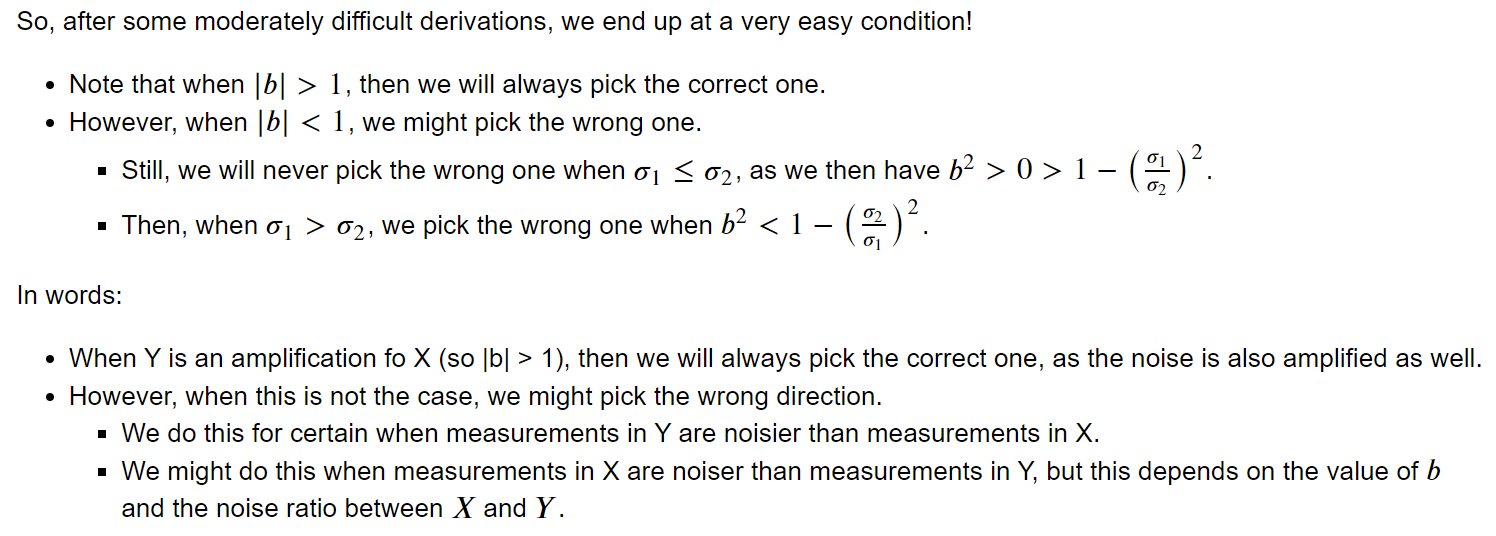
Setting the derivative to zero yields the solution.

1. **Which direction do OMP and NOTEARS pick?** Nevertheless, OMP and NOTEARS always ***seem*** to find the correct direction. How do they do it? By minimizing the mean squared error on the training data. This is the most sensible, but often not correct. So, *when picking an edge, OMP and NOTEARS always pick the direction that minimizes the MSE*.
2. **Based on b, σ1, and σ2, which direction do OMP and NOTEARS pick?**

The next question, **based on b, σ1, and σ2, which direction minimizes the MSE**? After some derivations, I determined that this is when

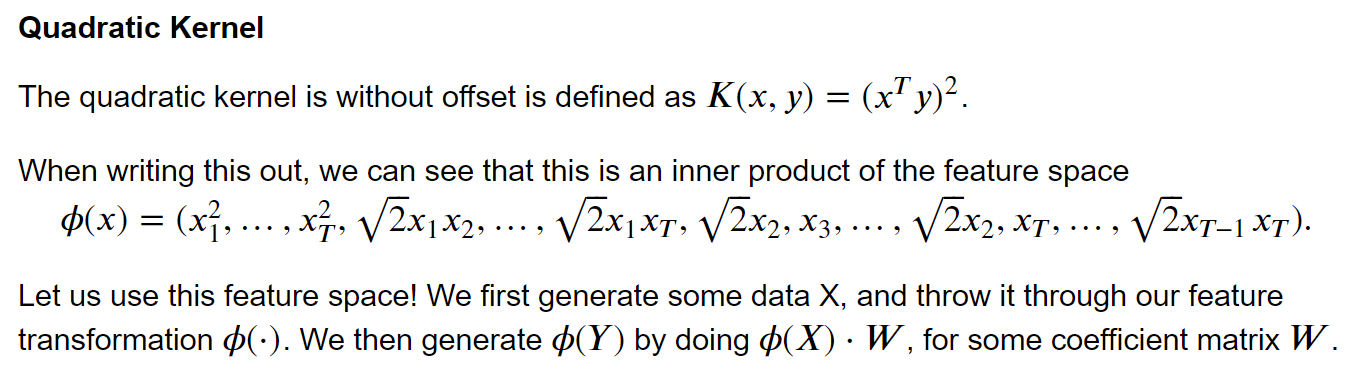


# 

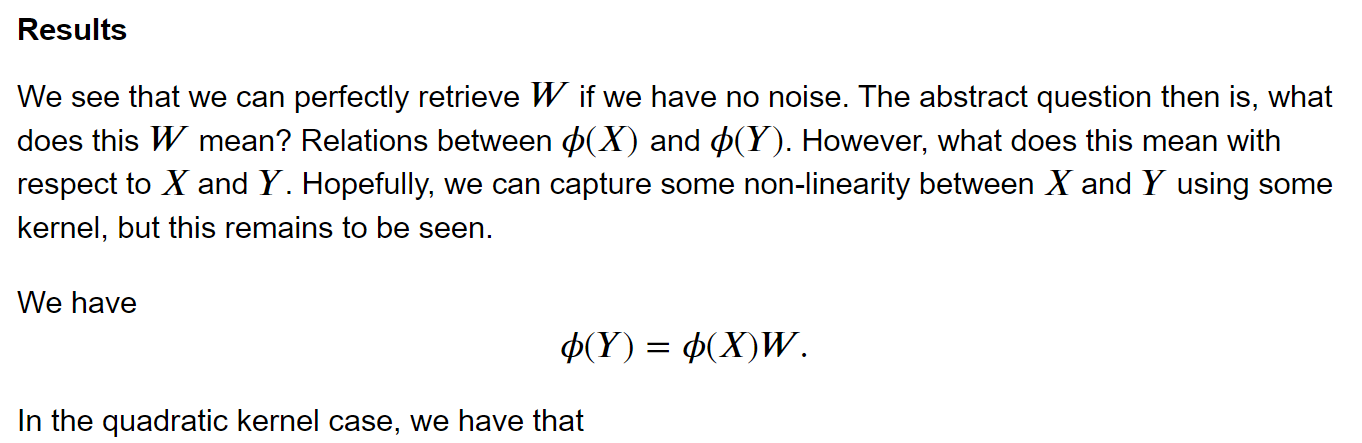


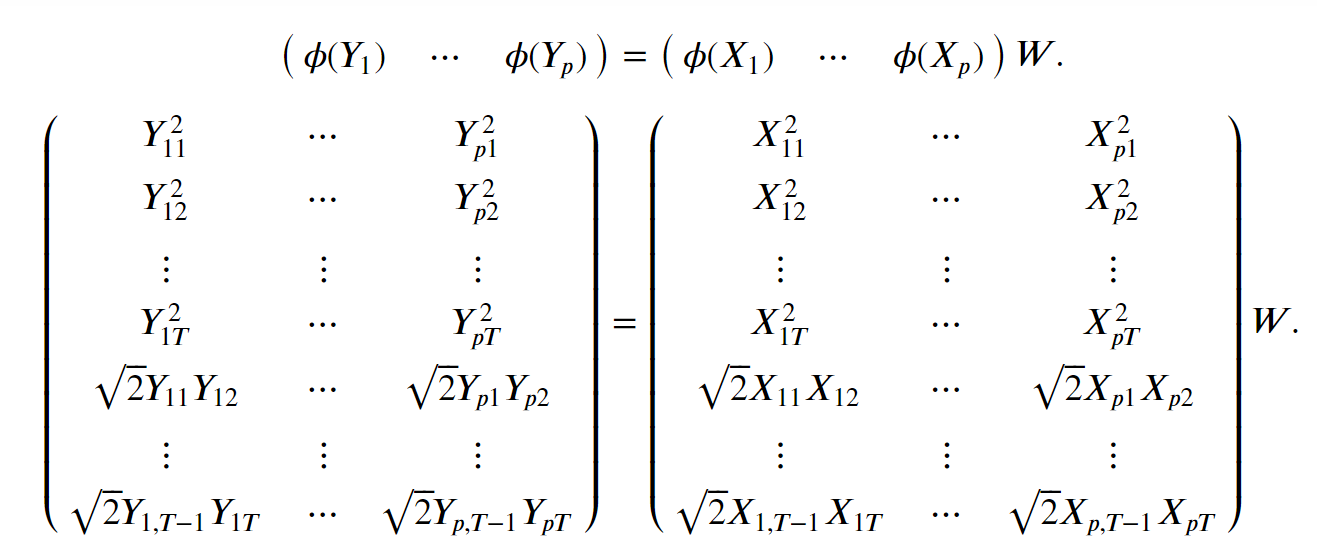
# Different Kernels for KOMP

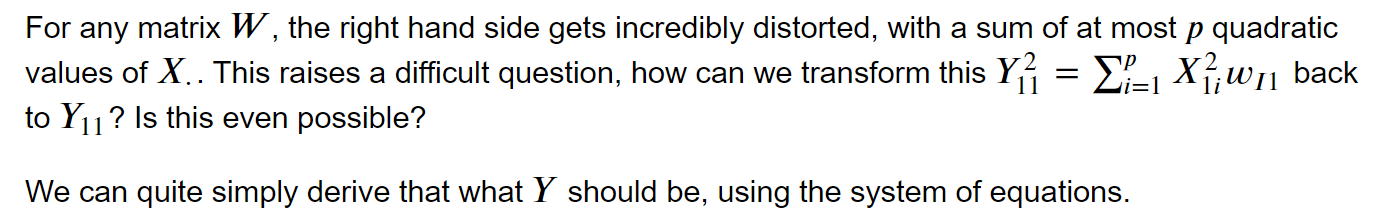
Reinvestigated Kernel-OMP further.



**Works, but only in the high-dimensional feature space**. When we add no noise, we can perfectly recover the true matrix W. This seems good, but we do not really know what the coefficients for W mean. Perhaps, if we can reconstruct Y, we can get more sense out of it.





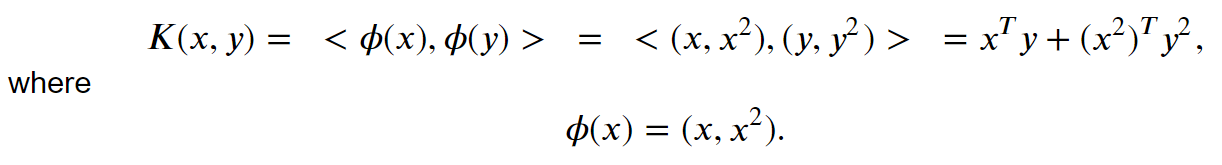


**Problem 1:** If coefficients in W are negative, then Y\_11^2 equals a negative number. How is this possible?

**Problem 2:** The system of equations has **no** solutions. Using Y\_11^2, Y\_12^2 and sqrt(2)Y\_11 Y\_12, we should be able to recover Y\_11 up to its sign. However, two different ways to determine the solution for Y\_1 do not agree, so it seems that we cannot recover Y and hence, not really say anything meaningful about this.

**“Own” Kernel**

I also tried a Kernel of my own, something simple with a linear and a quadratic term.



However, **same issues**. Works well with phi(Y) and phi(X), but as soon as you want to relate things to recover Y, things go wrong. We should simply be able to recover Y as the first half of phi(Y), but again this is too good to be true.

**RBF Kernel**

In the paper they seemed to use the RBF kernel on some Yale-B dataset for facial recognition, so there should be something possible. However, the Kernels do not seem to be doing what I want them to do, the translation back to the original feature space Y seems to be impossible.

# Backwards greedy OLS

Already implemented something like this, small discussion with Alex about it, which helped me on my way again. Quite similar, but not as fast, but seems to also find the most important edges. Also gives ordering of importance.

**OMP was forward greedy OLS.**

*While we can add edges:*

*Add the “best\*” suitable edge.*

*When we have a dense DAG, prune using e.g. bootstrapping, or cut-off using some tolerance.*

*“best\*”*: edge that is most correlated with the residual, so argmax <X[i], r>.

**Backwards Greedy OLS**

*Do OLS, get a full complete graph.*

*While we do not have a DAG:*

*Remove the “least important\*”* ***violating\*\**** *edge.*

*When we have a DAG, prune using e.g. bootstrapping, or cut-off using some tolerance*.

“*least important”*: edge that yields the smallest increase in MSE on training data.

**Violating**: edge that belongs to any cycle of length > 1 in the graph.

Some properties of Greedy OLS:

* Does not always find a full DAG. Often it is dense, but not full.
* Is quite slower than OMP, as gathering all violating edges can be relatively slow. Furthermore, verifying which of these edges is the least important also takes more time.

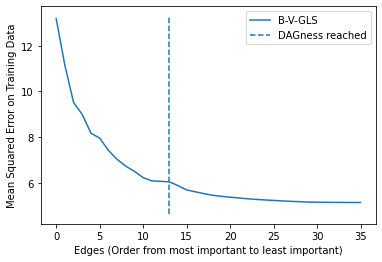
Some details of Greedy Backwards OLS

* How do you detect cycles? Quite easy to do in practice, but often also not very tractable. Simple algorithms exists that detect all cycle in polynomial time with respect to the number of cycles. However, the number of cycles can be exponential with respect to the number of nodes, so iterating over all cycles is not tractable in our scenario. The best way is to iterate over each edge, and check whether it is part of any cycle.
* How do you verify if an edge *w* is part of a cycle? An easy way is to compare the dag-ness criterion of the graph G(W) to the graph G(W \ w), so h(W \ w).- h(W) The value of h(W \ w) will always be equal to or lower than the value of h(W), as removing an edge only makes the graph more “DAG”-gy. So, a criterion could be h(W \ w) – h(W) > epsilon, for some very small value, like 10^-8.

Some issues

* Remove the “least important” “violating” edge is not very satisfying. As Alex remarked, why not just remove the least important edge, regardless is whether it is violating? This is however not very suitable, as this method will work poorly when there is one major model mismatch. For example, we start with a DAG, and add a large entry in the transpose position of the most important entry. Then, when these two entries are equally the most important, we will end up with a sparse matrix that only has one non-zero entry, as all other edges will have been removed before we remove these two. However, when there is no model mismatch, I expect this one to also recover the important edges.

**Results**



Backwards Greedy Least squares starts with a dense DAG (so 36 edges). From there, it first iteratively removes the least important edge that belongs to a cycle. We do this until we reach a DAG. This is when we have 13 edges (so we do not have a fully dense DAG of 21 edges). From there, we iteratively remove the least important edge.

From right to left, we see a clear climb until the vertical line. From there, we see that the MSE flattens a bit. This is because we do not remove the edges in order of importance, we first remove the violating edges. So, for edge 14 and 15, we remove the most important violating edge, and for edge 12 and 13, we remove the least important remaining non-violating edges. It makes sense that the most important violating edge is more important than the least important non-violating edge.

**Model Mismatch**

We now add one clear model mismatch. We use the matrix W:

[[ 0. 0. 0. 0. -0.64]

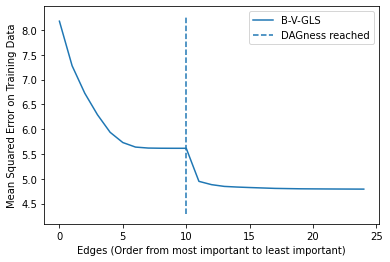
[ 0. 0. 0. 0. 0. ]

[-0.6 -0.46 0. 0. 0. ]

[ 0. 0.61 0. 0. 0. ]

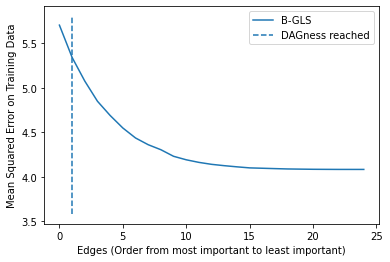
[-0.64 -0.5 0. 0. 0. ]]

We have a clear violation of the DAGness in the data generating matrix. How does B-V-GLS fare?

****

We see that we start with a dense DAG of 25 edges. From there, we remove all the least important violators until we have only one violation remaining, the underlined coefficients. Removing one of these results in a large jump in MSE, after which we have a DAG as you can see in the plot. From thereon, we continue by removing the least important edge iteratively.

What if we did not incorporate this “violators-first” policy? Then, we would get something like this:



We will end up with a “DAG” that only consists of one edge, which will most likely be one of the two underlined coefficients. As the two most important edges together create a cycle of length two, we will only have a DAG until we have one edge, which is when we have the dotted line at #edges = 1. This advocates for using the B-V-GLS over B-GLS.

**Methodologies**

I noticed that this method was comparable with OMP, but it differed in the selection criterion. I also recognizes LINGNAM-OLS in this. Therefore, I noticed that there were in fact three different selection criteria used, and also three different directions (ways to decide “next”).

**Directions:**

* *Forward*: Start with empty graph, iteratively add the next edge not violating DAGness.

Stop when gain is too low.

* *Backward*: Start with full graph, iteratively remove the next edge until DAG.

Possibly continue, until loss is too high.

* *Backward, violations first*: Start with full graph, iteratively remove next violating edge until DAG.

Possible continue, until loss is too high.

**Selection Criterion:**

* Correlation with residual:
  + Forward: add the edge with the largest correlation with residual: arg max <X\_i, r\_j>
  + *Backward: remove the edge with the smallest correlation with residual: arg max <X\_i, r\_j>*
* Mean Squared Error: arg min MSE(W \ wij) / arg min MSE(W u w\_ij)
  + Forward: add the edge yielding the largest decrease in MSE:
  + Backward: remove the edge yielding the smallest increase in MSE:
* Coefficient size: W: arg max / min {|w\_ij| | w\_ij =/= 0 }.
  + Forward: add the edge yielding the largest coefficient.
  + Backward: remove the edge with the smallest coefficient.

|  |  |  |  |
| --- | --- | --- | --- |
| Direction \ Criterion | Correlation with Residual | In / Decrease in MSE. | Coefficient size |
| Forward | **OMP** | **F-GLS** | LINGNAM-OLS |
| Backward | B-OMP | **B-GLS** | **LINGNAM-B-OLS** |
| Backward, violations first. | B-V-OMP | **B-V-GLS** | LINGNAM-B-V-OLS |