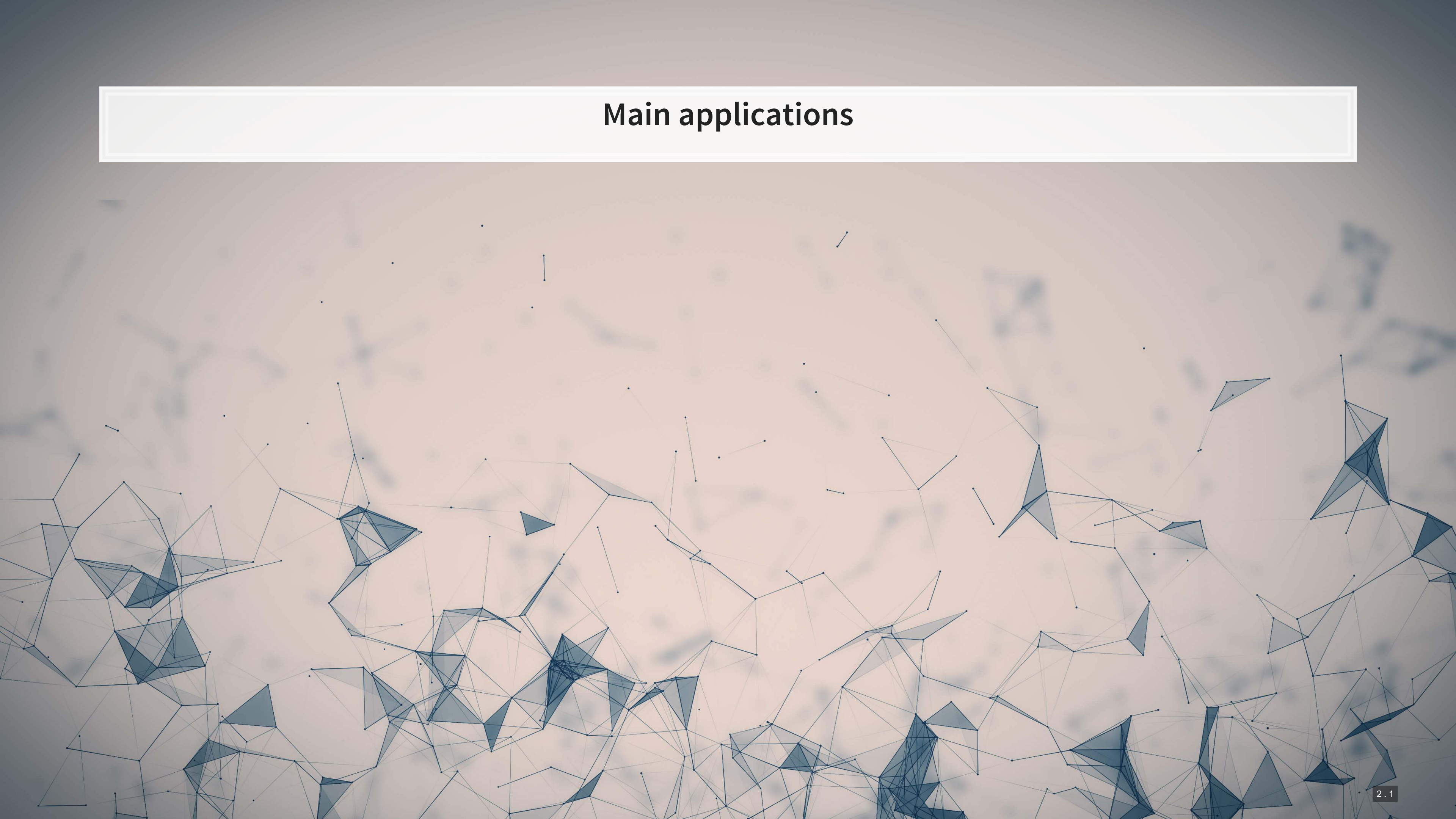


Session 2: Machine learning Drop-ins and Ensembling

2021 July 12

Dr. Richard M. Crowley
rcrowley@smu.edu.sg
<http://rmc.link/>

Main applications



Applications

Continue from Session 1

1. Predicting future stock return volatility based on 10-K filing discussion
 - We will apply SVR to this
2. Predicting 10-K/A irregularities using finance, textual style, and topics
 - We will apply SVM (SVC), XGBoost, and ensembling to this

Goals

1. Gain familiarity with using pure machine learning approaches
2. Try out a non-linear approach
3. Understand what ensembling is and why it is useful in *some* applications

Application 1: Linear problem (Recap)

- Idea: Discussion of risks, such as as foreign currency risks, operating risks, or legal risks should provide insight on the volatility of future outcomes for the firm.
- Testing: Predicting future stock return volatility based on 10-K filing discussion

Dependent Variable

- Future stock return volatility

Independent Variables

- A set of 31 measures of what was discussed in a firm's annual report

This test mirrors Bao and Datta (2014 MS)

Application 2: Binary problem (Recap)

- Idea: Using the same data as in Application 1, can we predict instances of intentional misreporting?
- Testing: Predicting 10-K/A irregularities using finance, textual style, and topics

Dependent Variable

Intentional misreporting as stated in 10-K/A filings

Independent Variables

- 17 Financial measures
- 20 Style characteristics
- 31 10-K discussion topics

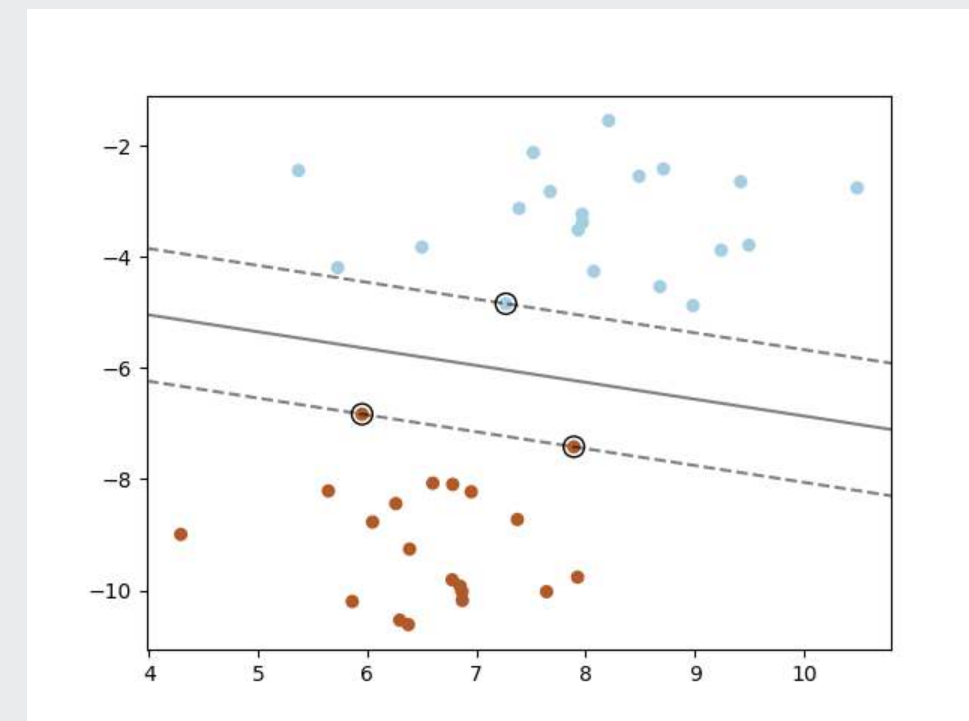
This test mirrors a subset of Brown, Crowley and Elliott (2020 JAR)

SVM: Support Vector Machine

What is SVM?

Simpler case: Binary Classification

- SVM-type algorithms generally focus on separability under some tolerance for error
 - This is quite different from our regression approaches
 - Regression focuses on *minimizing an error function*
- Note how in this example. the points that matter are those that are on the error boundaries
- The rest of the points aren't affecting the outcome much
 - You could shift them around on their respective side of the line with minimal impact



From the sklearn documentation

What are the benefits of SVM?

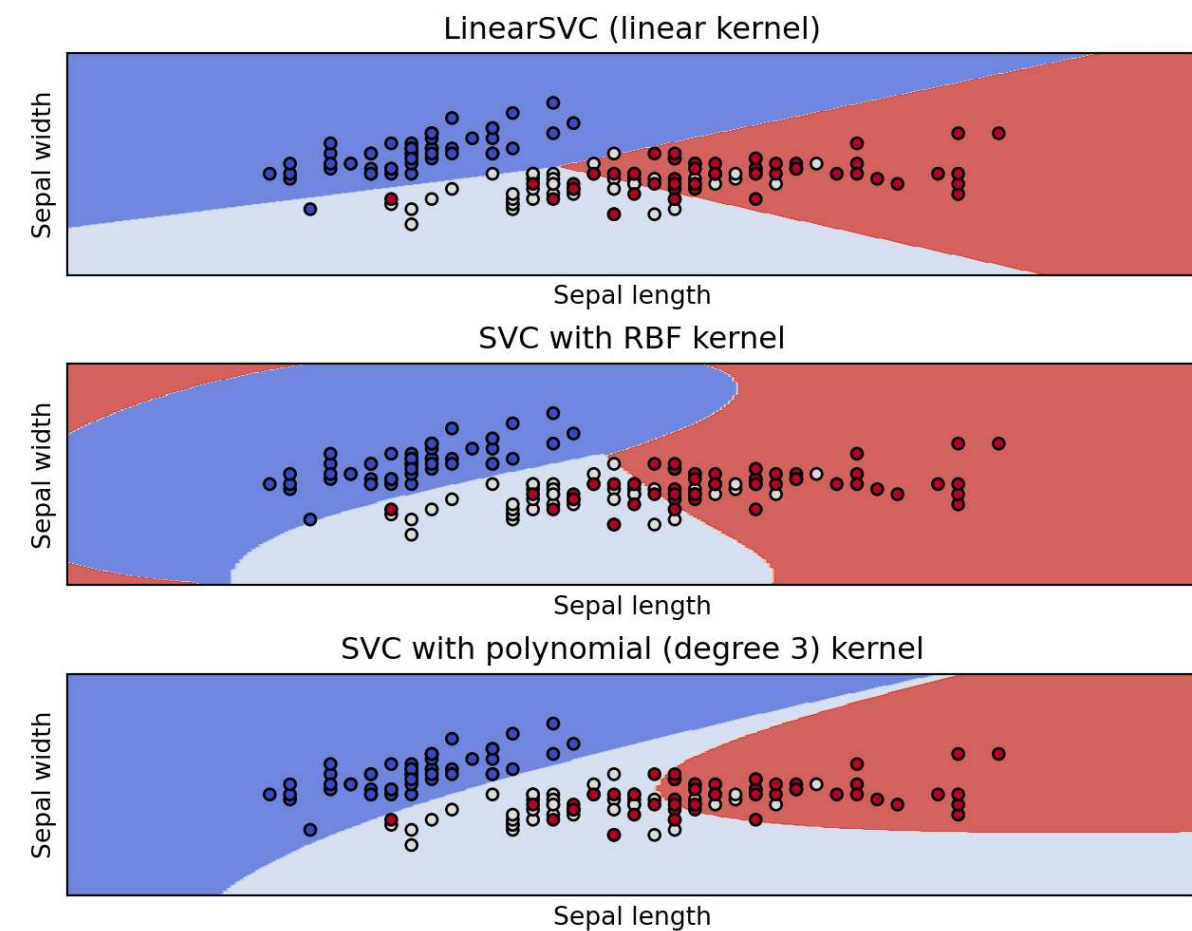
1. Non-linear kernels

- SVM can be linear or non-linear
- 3 examples to the right, [adapted from the sklearn documentation](#)

2. Different objective function than regression

- Fits better with classification, conceptually

3. Can work with non-numeric data (text, images, graphs)



What are the costs of SVM?

1. Doesn't work well on noisy data
2. Can be slow to train on datasets with many observations
3. Difficult to interpret model when using a non-linear classifier
4. Can be difficult to pick an optimal kernel

Implementing SVM in python

- For this we will use `sklearn` again
- To keep things simple and interpretable, we will use linear kernels in these examples

Binary classification

- Fast linear model:
 - `sklearn.svm.LinearSVC()`
- General model:
 - `sklearn.svm.SVC()`

Regression

- Fast linear model:
 - `sklearn.svm.LinearSVR()`
- General model:
 - `sklearn.svm.SVR()`

- Both linear methods have a hyperparameter C which controls the amount of regularization (inversely)
 - We can tune this using `sklearn` as well!

Why are there two ways each to run a linear SVM model?

- The two ways use different backends
 - The `LinearSV_` methods use a backend called `liblinear`
 - The `SV_` methods use a backend called `libsvm`
- Liblinear is faster but only supports linear kernels
 - Time to run is roughly linear in the number of observations
 - Libsvm is fast on small samples, but time increase for additional observations is polynomial
- The results aren't quite the same across backends
 - Liblinear uses a penalized intercept
 - Liblinear optimizes a “squared hinge” loss function
 - Libsvm optimizes “hinge” loss

$$\text{hinge}(x, y) = \max(0, 1 - y \cdot f(x)), \quad y \in \{-1, +1\}, \quad f(x) \in \mathbb{R}$$

Both developed out of National Taiwan University, and both maintained by the same professor

Implementing LinearSVC for irregularity detection

- To train a simple linear SVM classifier, we can call `svm.LinearSVC()` pretty much the same way that we used `linear_model.Lasso()` earlier
 - Note: The `dual=False` option is to maintain efficiency when the number of observations is great than the number of variables

```
model_svc = svm.LinearSVC(C=1, dual=False)
model_svc.fit(train_X_logistic, train_Y_logistic)
```



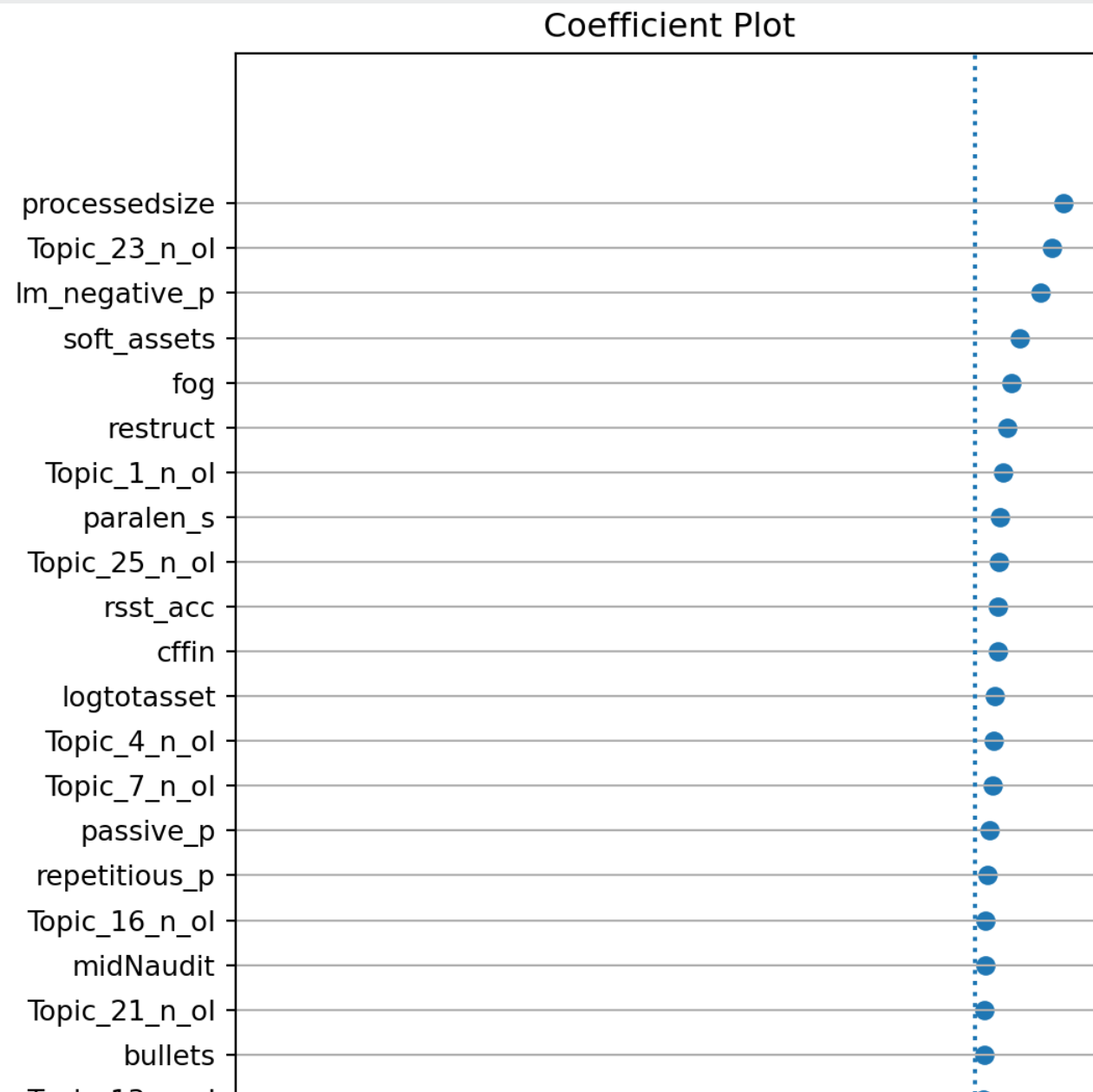
- No regression table built in, but we can visualize it with `coefplot()`

```
coefplot(vars_logistic, model_svc.coef_)
```

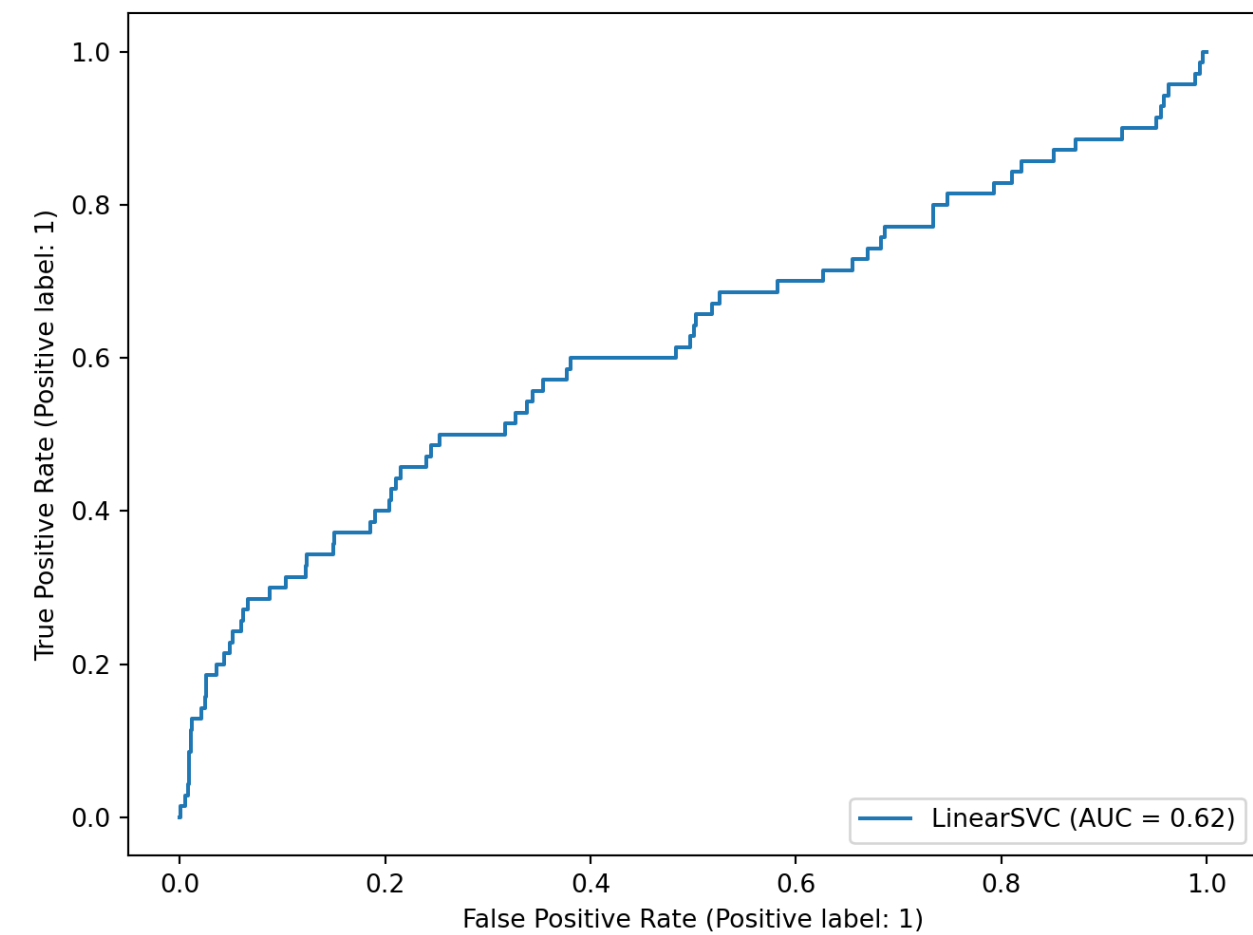


Visualizing LinearSVC for irregularity detection

```
coefplot(vars_logistic, model_svc.coef_)
```



```
metrics.plot_roc_curve(model_svc, test_X_logistic,  
                        test_Y_logistic)
```



Optimizing the C parameter

```
C_range = np.logspace(-2, 6, 9)
param_grid = dict(C=C_range)
cv = model_selection.StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=1)
grid_svc = model_selection.GridSearchCV(svm.LinearSVC(dual=False), param_grid=param_grid, cv=cv)
grid_svc.fit(train_X_logistic, train_Y_logistic)
```

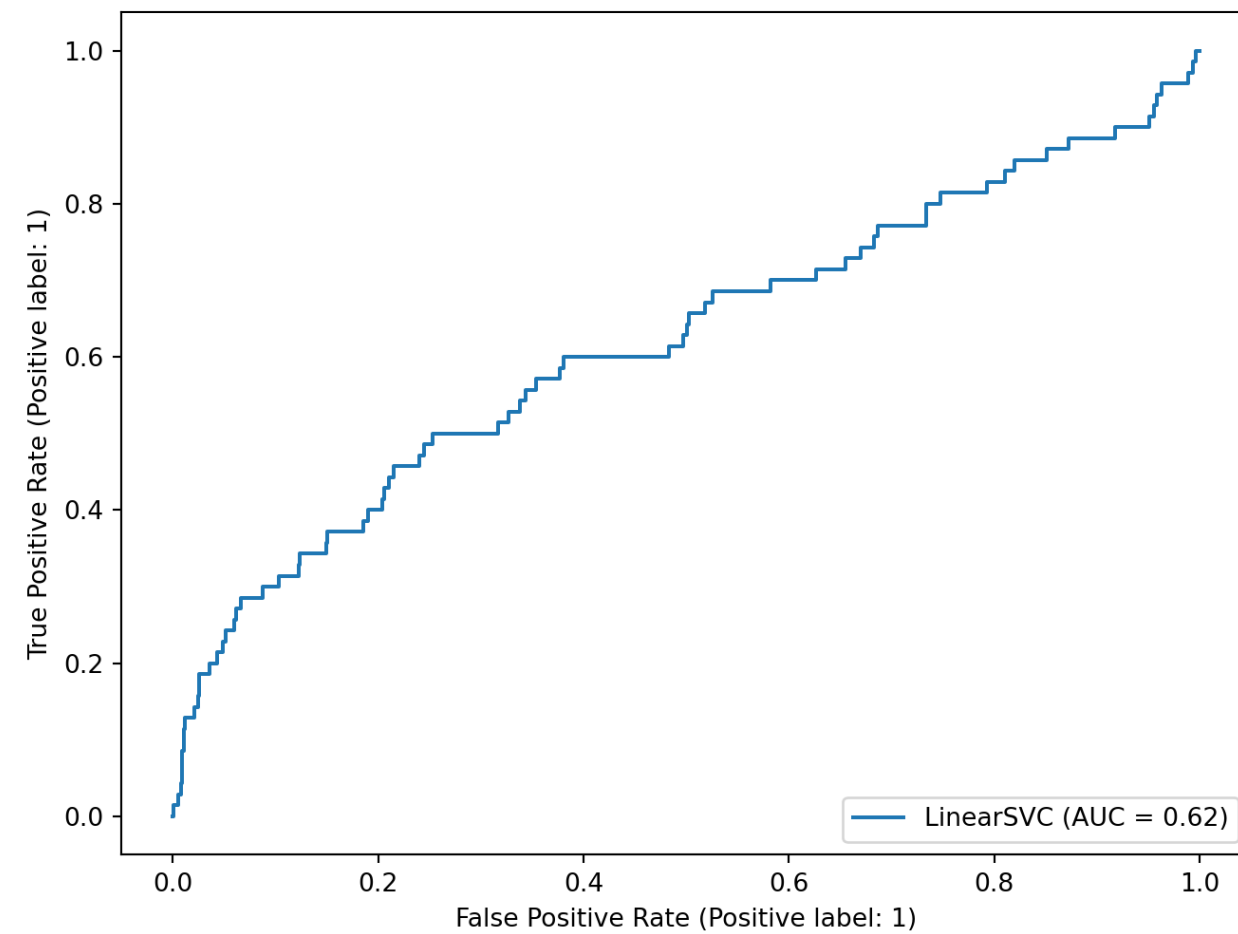
```
## GridSearchCV(cv=StratifiedShuffleSplit(n_splits=5, random_state=1, test_size=0.2,
##      train_size=None),
##      estimator=LinearSVC(dual=False),
##      param_grid={'C': array([1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02, 1.e+03, 1.e+04, 1.e+05,
##      1.e+06])})
```

```
print("The best parameters are %s with a score of %0.2f"
      % (grid_svc.best_params_, grid_svc.best_score_))
```


Comparison pre- vs post-optimization: ROC

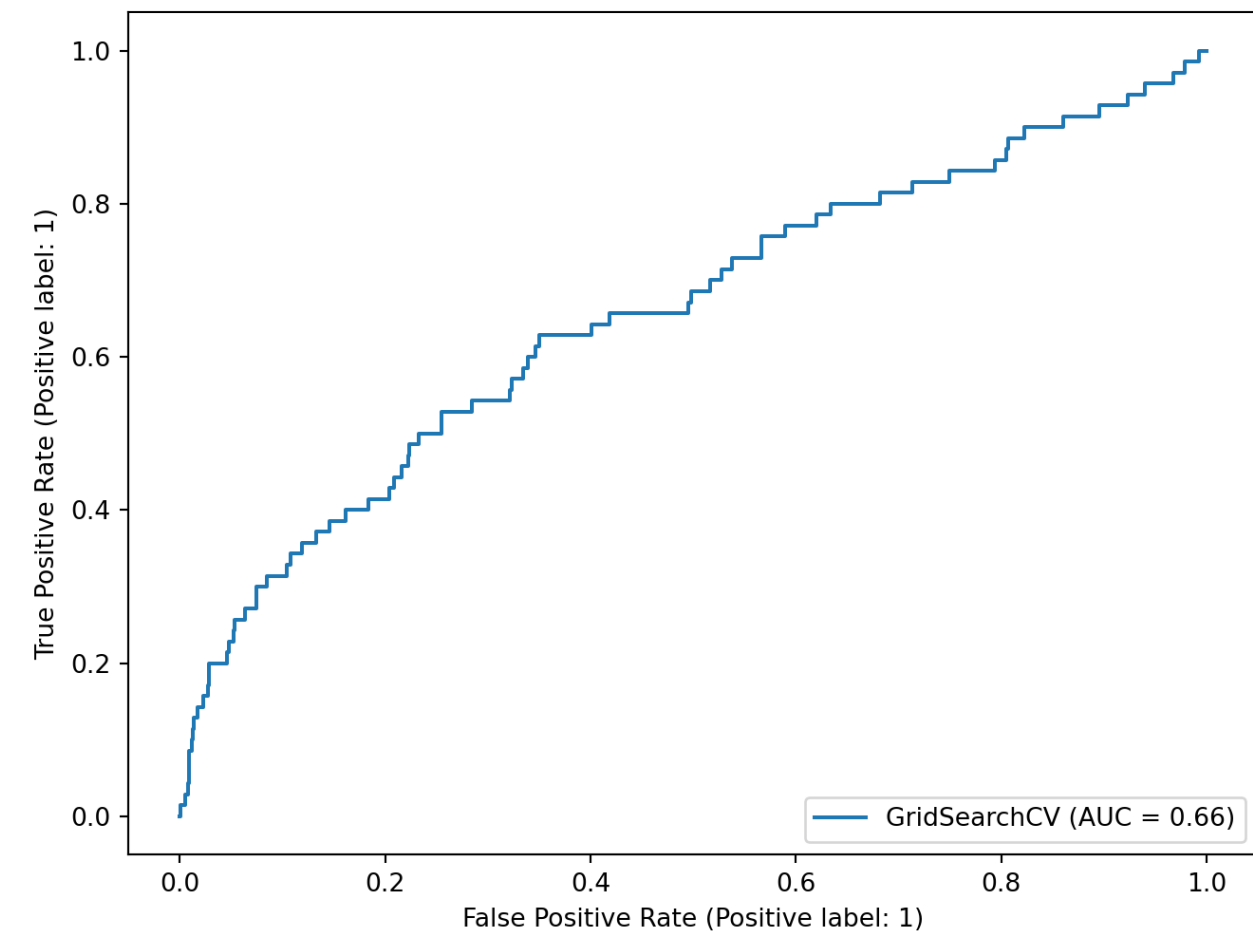
Unoptimized

```
metrics.plot_roc_curve(model_svc, test_X_logistic,  
                        test_Y_logistic)
```



Optimized

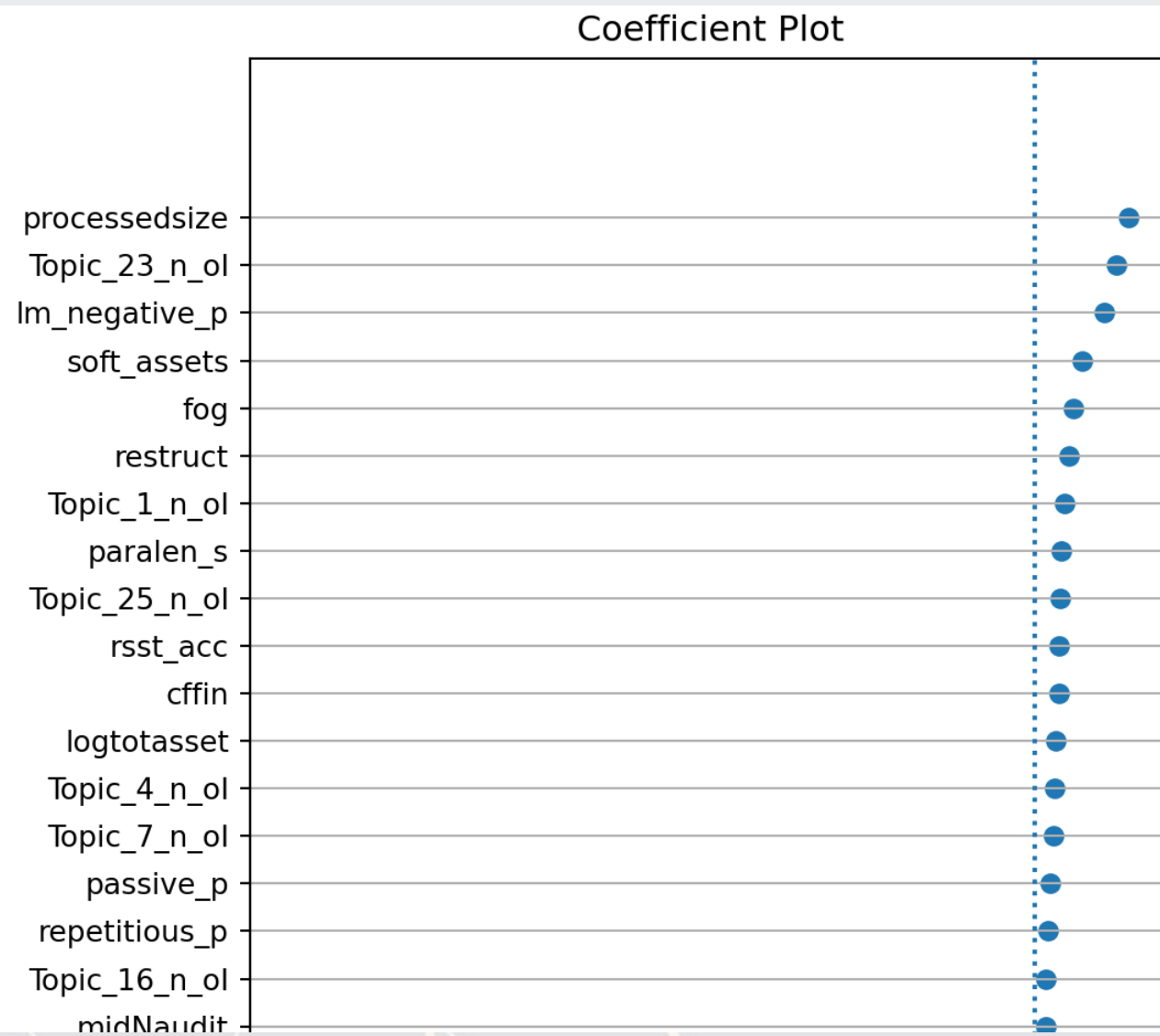
```
metrics.plot_roc_curve(grid_svc, test_X_logistic,  
                        test_Y_logistic)
```



Comparison pre- vs post-optimization: Coefficients

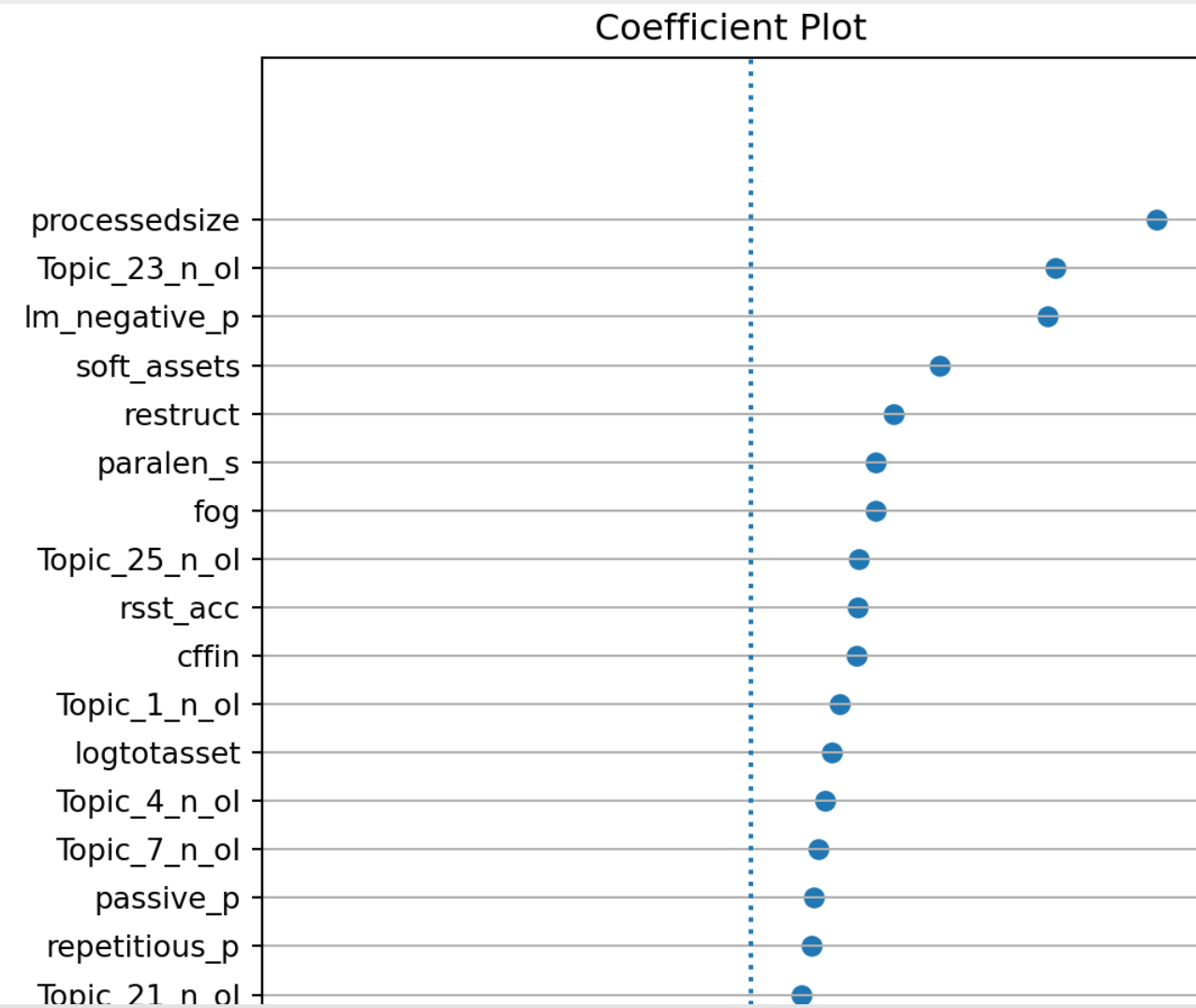
Unoptimized

```
coefplot(vars_logistic, model_svc.coef_)
```



Optimized

```
coefplot(vars_logistic,  
grid_svc.best_estimator_.coef_)
```



Visualizing with UMAP

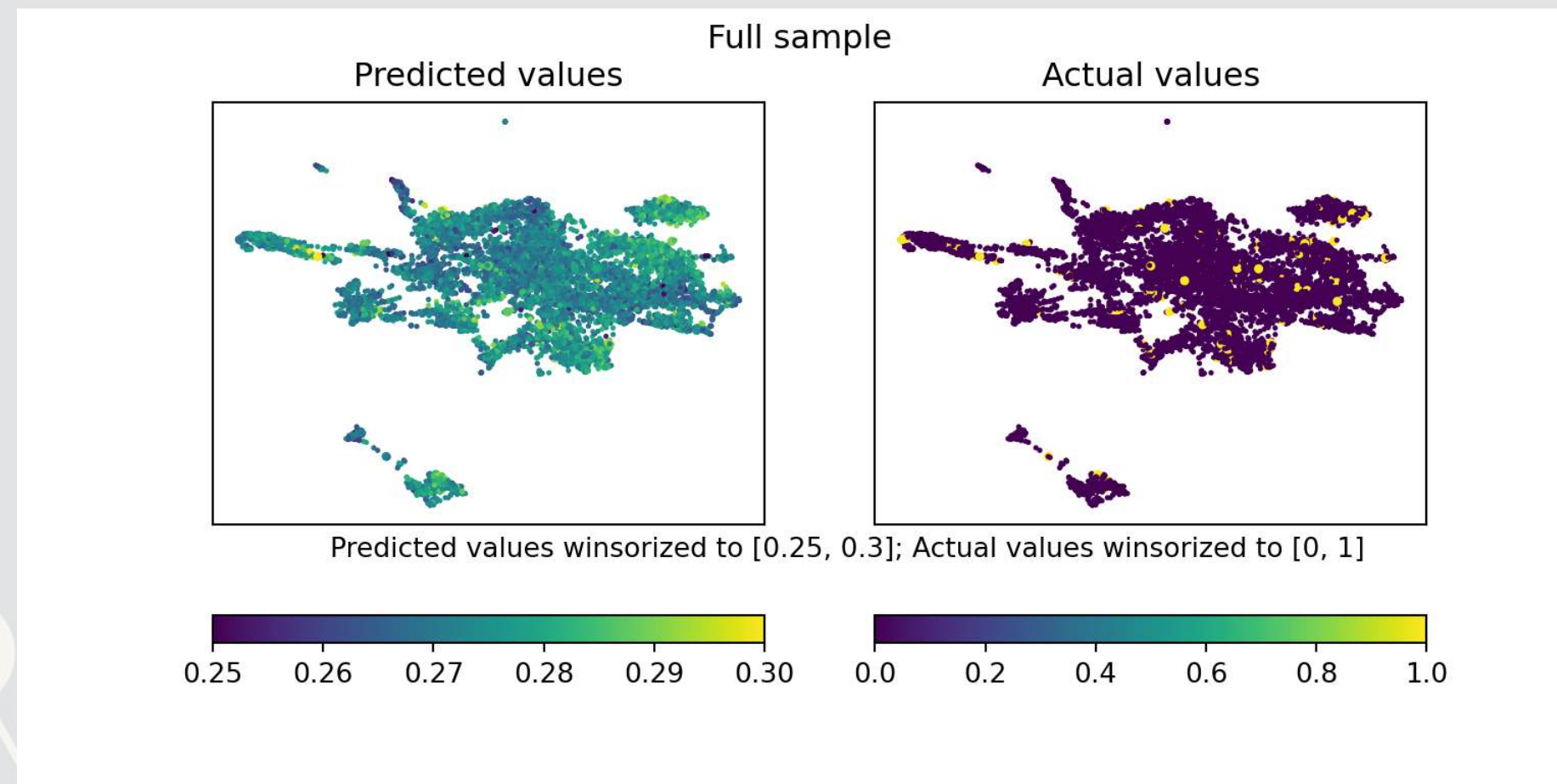
What is UMAP?

- UMAP stands for Uniform Manifold Approximation and Projection for Dimension Reduction
 - From Leland, Healy and Melville (2018) (2k+ cites already)
- It is useful for dimensionality reduction, like PCA
 - We will use it to reduce 68 dimensions down to 2
- It is useful for plotting 2 dimensional representations of high dimensional data by maintaining local distance structures, like t-SNE
 - Unlike t-SNE, it is efficient to run

UMAP essentially uses Riemannian manifolds and tries to maintain geodesic distance around a point – it is well supported theoretically

Visualizing what SVM is doing using UMAP

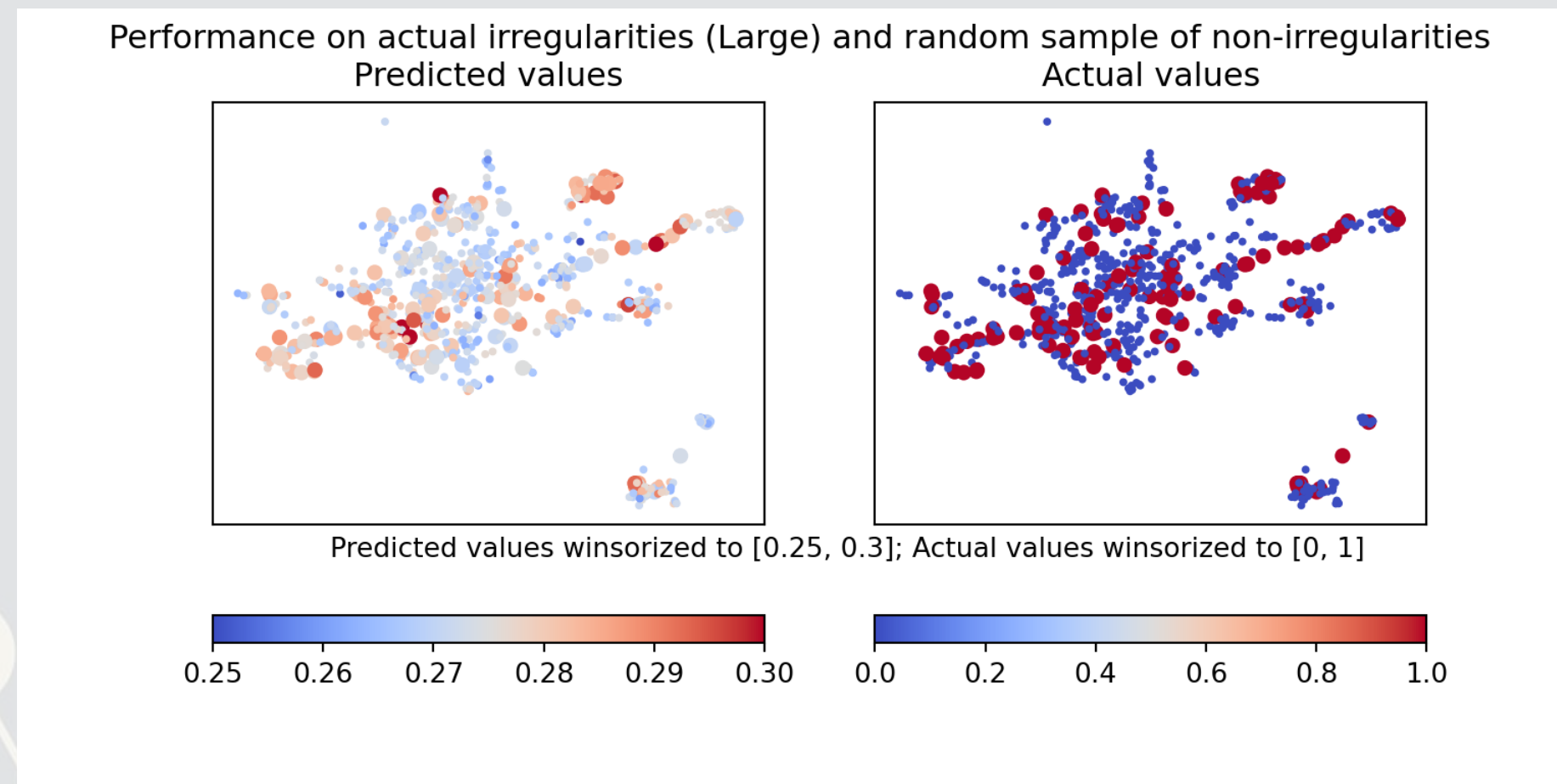
```
train_Yhat_logistic = logistic(grid_svc.decision_function(train_X_logistic))
umap_compare_svm(train_X_logistic, train_Yhat_logistic, train_Y_logistic,
                 clip=[[0.25, 0.3], [0, 1]], binary=5, title="Full sample")
```



The data is really noisy

Visualizing what SVM is doing using UMAP

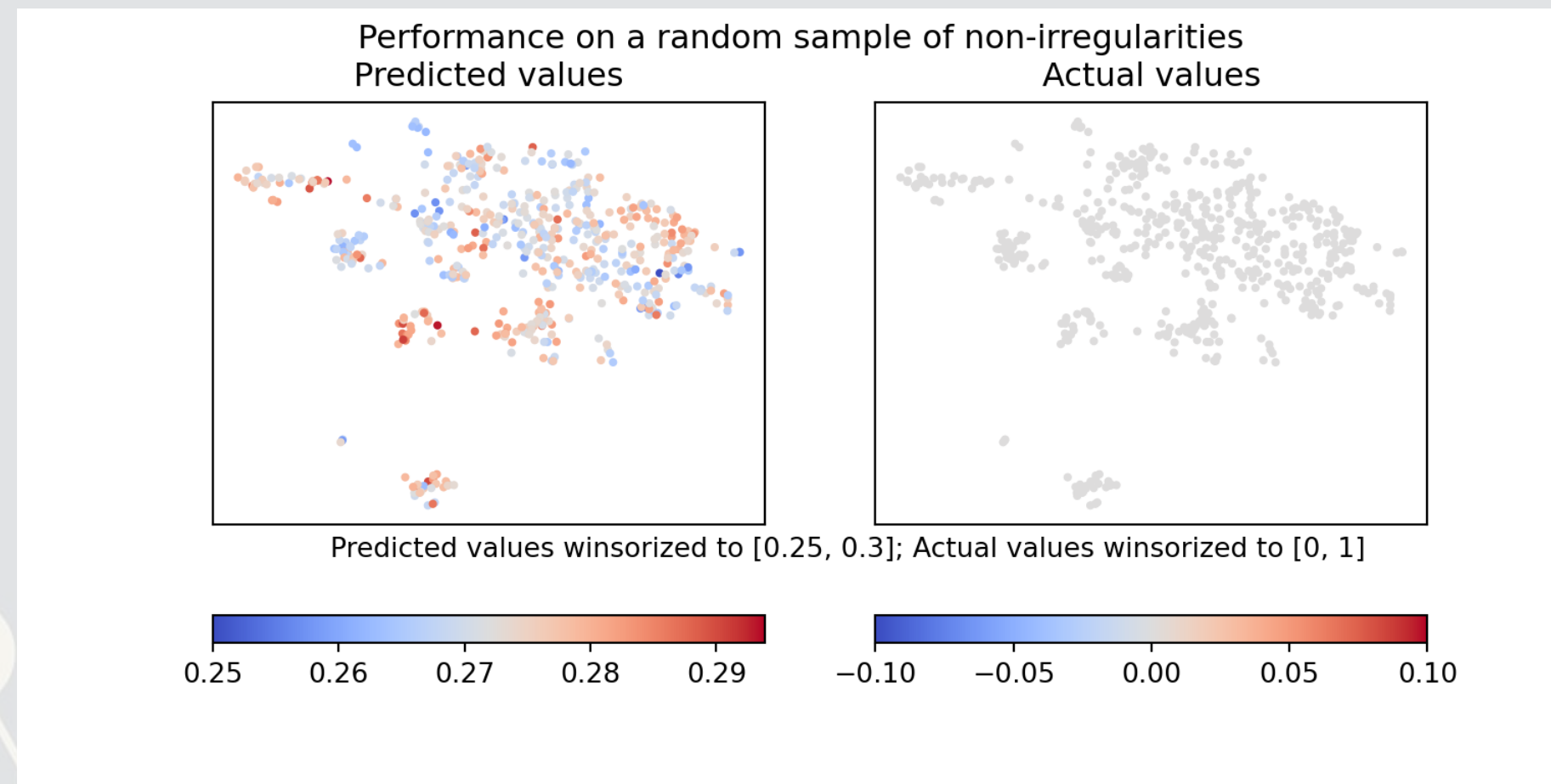
```
umap_compare_svm(train_X_logistic, train_Yhat_logistic, train_Y_logistic, clip=[[0.25, 0.3], [0, 1]], cmap='coolwarm', binary=True, subset=((train_Y_logistic==1) | (np.random.rand(len(train_Y_logistic))<0.05)), title="Performance on actual irregularities (Large) and random sample of non-irregularities")
```



Type I errors are pretty minimal – the algorithm is rarely very off

Visualizing what SVM is doing using UMAP

```
umap_compare_svm(train_X_logistic, train_Yhat_logistic, train_Y_logistic, clip=[[0.25, 0.3], [0, 1]], cmap='coolwarm', binary=True, subset=((train_Y_logistic==0) & (np.random.rand(len(train_Y_logistic))<0.05)), title="Performance on a random sample of non-irregularities")
```



There are definitely some combinations of parameters that are consistently leading to Type II errors

SVM for regression: SVR

```
model_svr = svm.LinearSVR(C=1, dual=False,  
                           loss='squared_epsilon_insensitive')  
model_svr.fit(train_X_linear, np.ravel(train_Y_linear))
```



```
C_range = np.logspace(-4, 6, 11)  
param_grid = dict(C=C_range)  
cv = model_selection.KFold(n_splits=5)  
grid_svr = model_selection.GridSearchCV(  
    svm.LinearSVR(dual=False,  
                  loss="squared_epsilon_insensitive"),  
    param_grid=param_grid, cv=cv)  
grid_svr.fit(train_X_linear, np.ravel(train_Y_linear))
```





```
## GridSearchCV(cv=KFold(n_splits=5, random_state=None, s  
##             estimator=LinearSVR(dual=False,  
##                                 loss='squared_epsilon  
##             param_grid={'C': array([1.e-04, 1.e-03, 1  
##             1.e+04, 1.e+05, 1.e+06]))
```

```
print("The best parameters are %s with a score of %0.2f"  
      % (grid_svr.best_params_, grid_svr.best_score_))
```



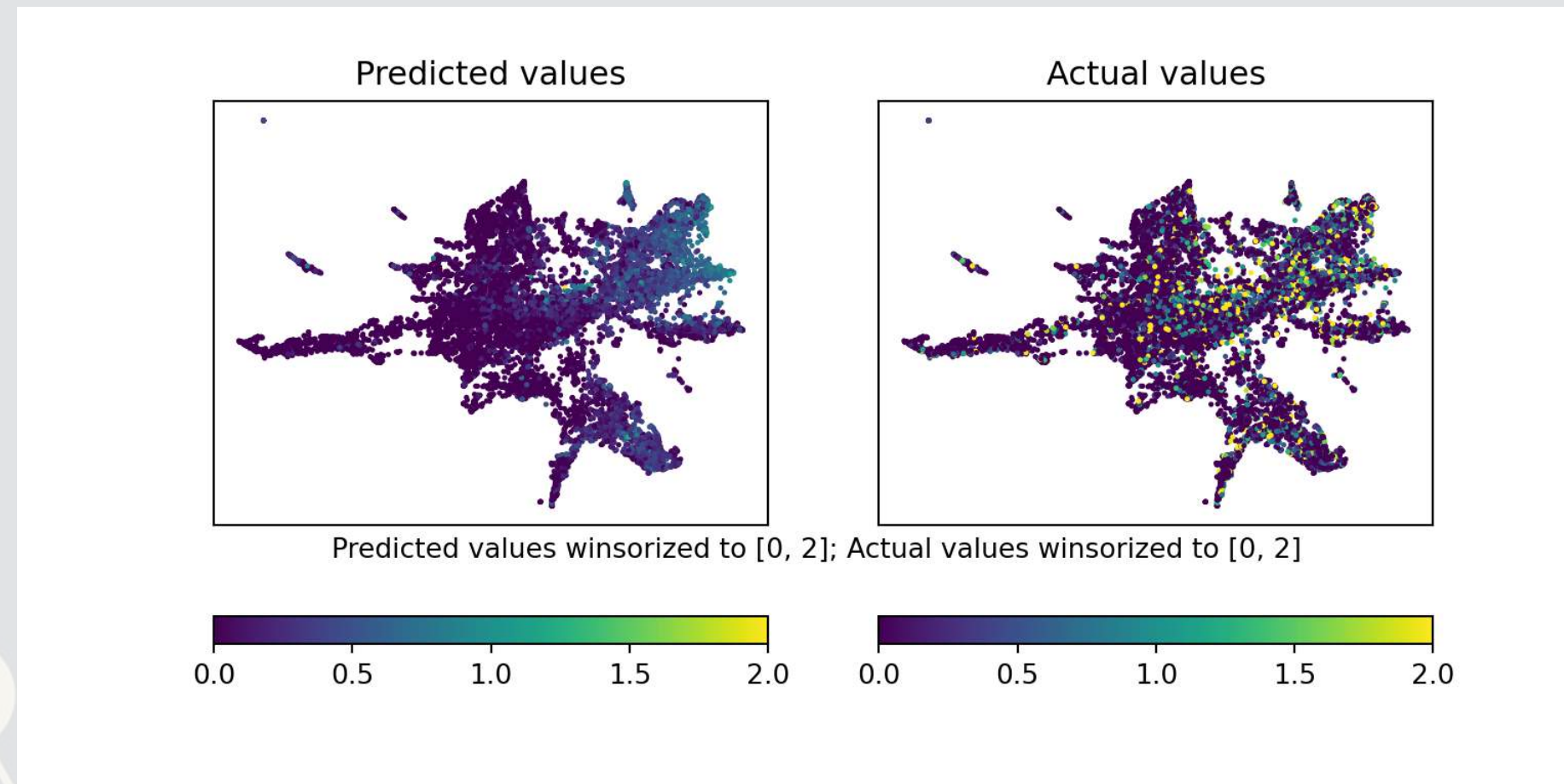
SVR coefficients

```
coefplot(vars_linear, model_svr.coef_) 
```

```
coefplot(vars_linear, grid_svr.best_estimator_.coef_) 
```


Visualizing SVR with UMAP

```
train_Yhat_linear = model_svr.predict(train_X_linear)  
umap_compare_svm(train_X_linear, train_Yhat_linear, train_Y_linear, clip=[[0, 2], [0, 2]])
```



Here we see some clusters that are indeed higher in volatility being picked up correctly by SVM

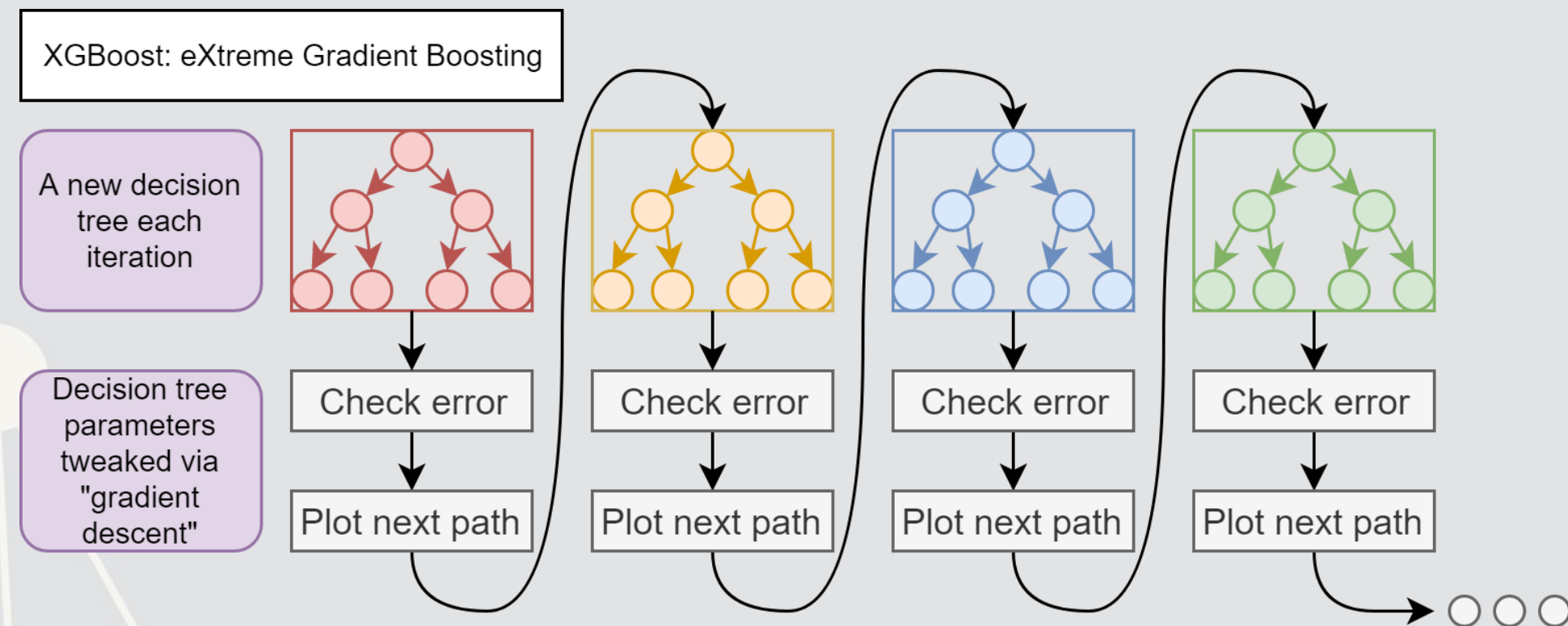
Addendum: Using R

- SVM is easy to implement using the `svm()` function from `e1071`
- The `kernlab` package also implements SVM, focusing largely on kern-based ML algorithms

xGBoost: Extreme Gradient Boosting

What is XGBoost

- eXtreme Gradient Boosting
- A simple explanation:
 1. Start with 1 or more decision trees & check error
 2. Make more decision trees & check error
 3. Use the difference in error to guess a another model
 4. Repeat #2 and #3 until the model's error is stable



XGBoost: Foundations

- XGBoost has its roots in AdaBoost (Adaptive Boosting)
 - Adaboost uses a sequence of weak learners to build a model
 - Combats against overfitting, and the sequence of individually weak models converges to be a strong learner
 - The convergence part is mathematically proven!
 - XGBoost isn't as theoretically founded as Adaboost'
 - It trades off some mathematical rigor for flexibility and empirical performance

Benefits of XGBoost

- Tree based
 - Inherently non-parametric (no assumptions on data distribution)
- Non-linear but still somewhat interpretable
- Robust to noise
- Can handle missing or categorical variables
- Robust to overfitting (somewhat)

As compared to other tree algorithms

- Implements gradient descent to sequentially grow trees
- Parallelizable (so it can be computed efficiently)
- Supports regularization

Drawbacks of XGBoost

So

many

hyperparameters.

- This makes it difficult to train a model well
 - But it is hard to beat a well trained XGBoost model with anything else we have discussed thus far
- It may technically be interpretable, but interpreting a big model is still difficult
- Like most tree-based methods, it struggles with extrapolation that is outside the bounds of its input data.

XGBoost in economics

Deryugina, Heutal, Miller, Molitor and Reif (2019 AER)

- What is the problem?
 - Predicting one-day mortality to show whether air pollution leads to more deaths in vulnerable populations
 - The focus of the papers is largely on causality
- The treatment is based on *wind direction*
 - Wind direction associated with above median pollution conditional on fixed effects
 - County, year-month, and state-month FEs

Why use XGBoost here?

- To “thoroughly investigate heterogeneity in vulnerability to dying from acute air pollution exposure”
- Following a particular method from Chernozhukov, Demirer, Duflo, and Fernandez-Val (2018)
 - We won’t focus on the method today, just the XGBoost implementation
 - We *will* focus on the general method in Session 5
 - Technically, the CDDF methodology works with any ML classifier
 - Cites Einav et al. (2018 Science) on the efficacy of XGBoost for mortality for medicare populations
- 2 issues
 1. Low probability of dying on any given day
 - Solution: Downsample the data to artificially increase the probability of dying
 2. 20 billion observations in the sample...
 - Simpler fixed effects + partitioning the data (250 times) + averaging
 - This is solvable now in XGBoost with larger than memory computation

Results

TABLE 6—BEST LINEAR PREDICTION OF THE CONDITIONAL AVERAGE TREATMENT EFFECT

Parameter	(1)	(2)	(3)	(4)
β_1 (average treatment effect)	1.15 (0.221)	1.06 (0.200)	1.08 (0.263)	1.11 (0.247)
β_2 (heterogeneity)	0.0148 (0.003)	0.0133 (0.00268)	0.0137 (0.00346)	0.0126 (0.00317)
Horvitz-Thompson transformation		X		X
Trimming threshold	5%	5%	1%	1%
Observations (millions)	21,724	21,724	23,075	23,075

Notes: Columns 1 and 3 present regression estimates of equation (5) from the main text. Columns 2 and 4 present estimates of equation (A6) from the online Appendix. The dependent variable is the one-day mortality rate per million beneficiaries. The parameter β_1 measures the average mortality effect of being exposed to one day of air when the wind that day is blowing from a direction associated with high air pollution. Rejecting the null hypothesis that $\beta_2 = 0$ implies that heterogeneity is present and that the proxy predictor, $\hat{S}(Z_{it})$, captures a component of this heterogeneity. These regressions omit observations with estimated propensity scores less than the trimming threshold or greater than 1 minus the threshold. Standard errors, clustered by county, are reported in parentheses.

Implementing XGBoost in python: Setup

- The `xgboost` package has two different interfaces for running models
- The default version (which invokes `xgb.train()`) needs us to convert data

```
dtrain = xgb.DMatrix(train_X_logistic, label=train_Y_logistic, feature_names=vars_logistic)
dtest = xgb.DMatrix(test_X_logistic, label=test_Y_logistic, feature_names=vars_logistic)
```



- There is also an `sklearn` compatible version that is useful for running cross validation
 - This uses the same matrices we used for SVM as input

XGBoost parameters

```
param = {  
    'booster': 'gbtree',           # default -- tree based  
    'nthread': 8,                 # number of threads to use for parallel processing  
    'objective': 'binary:logistic', # binary, output probabilities  
    'eval_metric': 'auc',         # maximize ROC AUC  
    'eta': 0.3,                   # shrinkage; [0, 1], default 0.3  
    'max_depth': 6,               # maximum depth of each tree; default 6  
    'gamma': 0.1,                 # set above 0 to prune trees, [0, inf], default 0  
    'min_child_weight': 1,        # higher leads to more pruning of tress, [0, inf], default 1  
    'subsample': 0.8,             # Randomly subsample rows if in (0, 1), default 1  
    'colsample_bytree': 0.8,      # Randomly subsample variables if in (0, 1), default 1  
    'random_state': 70  
}  
num_round = 30
```



Running XGBoost

- We use `xgb.train()` to fit the model

```
model_xgb_logistic = xgb.train(param, dtrain, num_round)
```



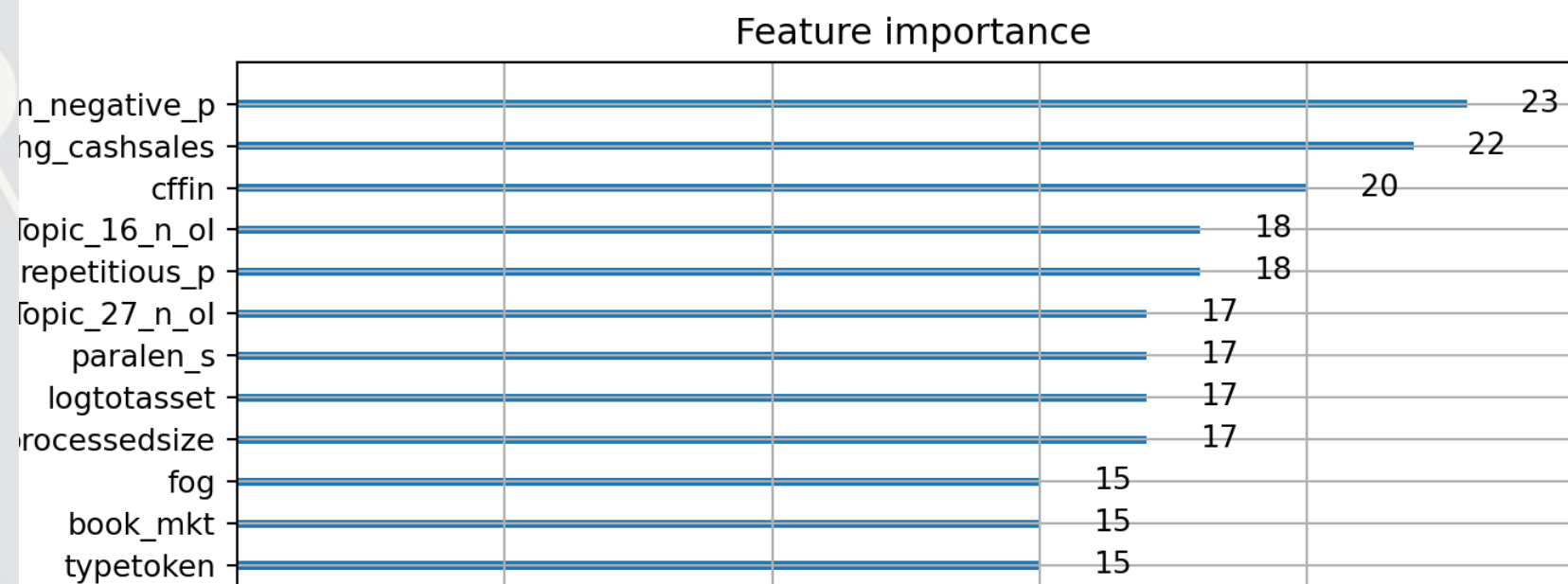
```
test_Yhat_xgb_logistic = model_xgb_logistic.predict(dtest)
auc = metrics.roc_auc_score(test_Y_logistic, test_Yhat_xgb_logistic)
print('AUC is {}'.format(auc))
```

```
fpr, tpr, thresholds = metrics.roc_curve(test_Y_logistic, test_Yhat_xgb_logistic)
display = metrics.RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=auc)
display.plot()
```

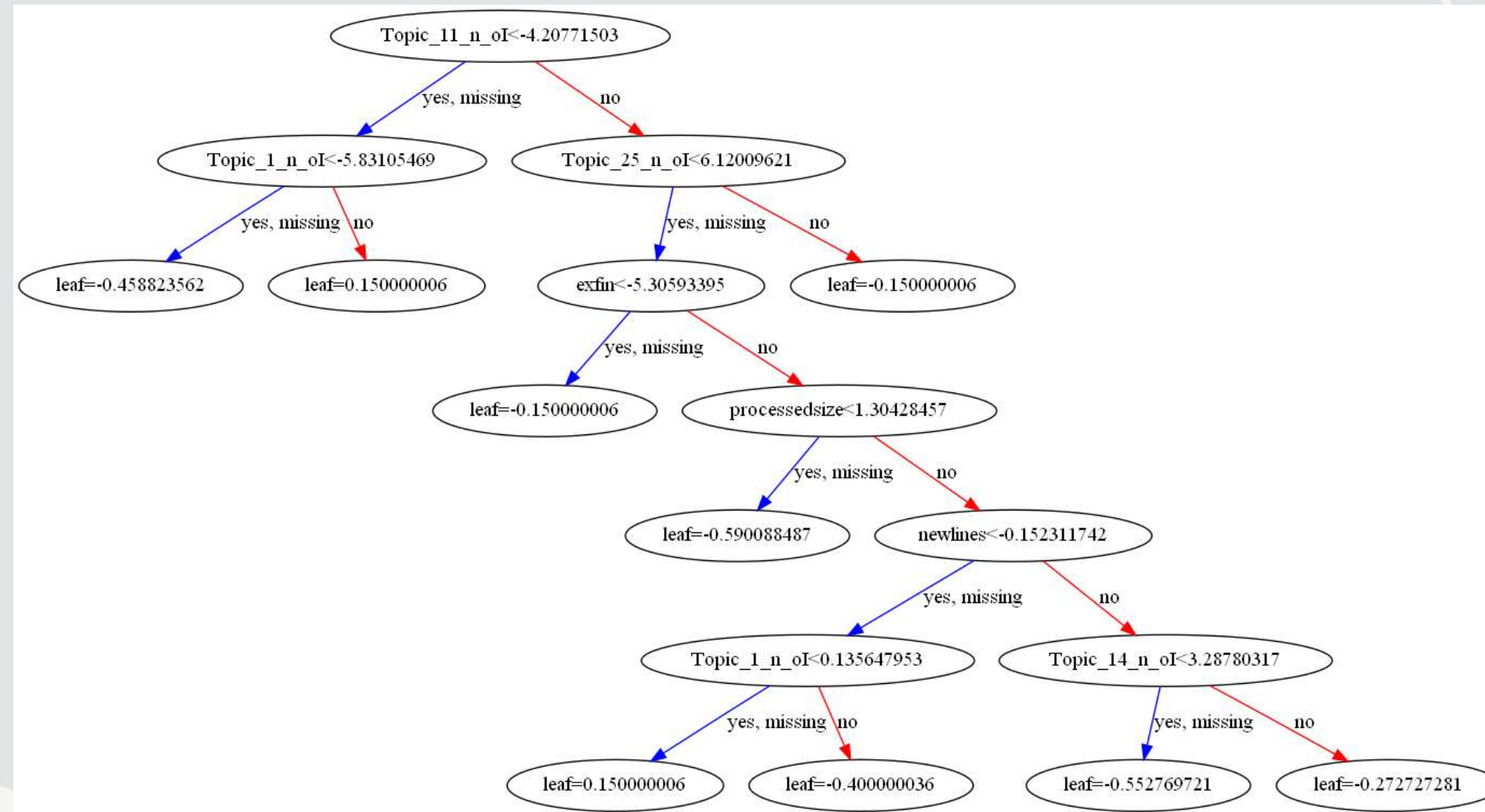

Analyzing the model: Importance plot

- The importance plot shows which variables have the greatest impact on the model
 - A higher number = more important
- In this case, we see a mix of sentiment, financial, topic, and grammatical measures in the top 5 measures

```
fig, ax = plt.subplots(figsize=(8,16))  
xgb.plot_importance(model_xgb_logistic, ax=ax)
```



Analyzing the model: Seeing the trees



One of 30 trees in the model

Aside: Exporting trees

If you have `graphviz` installed, you can use that to export pictures of each tree in the model

```
# If you want to view every tree contained in your final model, the below code will dump a PNG file of each tree  
# into a "trees/" directory in the same folder as this file.  
num_trees = len(model_xgb_logistic.get_dump())  
for tree_index in range(0, num_trees):  
    dot = xgb.to_graphviz(model_xgb_logistic, num_trees=tree_index)  
    dot.format = 'png'  
    dot.render("xgb_trees/tree{}".format(tree_index))
```

Aside: Parameter optimization

- Parameter optimization for a model with so many parameters is tricky
- A fully worked out version is included in the jupyter notebook for this session
 - It starts from the same parameters we used for our model
 - It then does a grid search to optimize:
 - `max_depth` and `min_child_weight`
 - `max_depth` and `min_child_weight`
 - `eta`
 - `gamma`
 - `subsample` and `colsample_bytree`
 - `subsample` and `colsample_bytree`
 - Number of rounds

The example is less exhaustive than you should be for a research paper – use finer grids, but it will take much longer to run

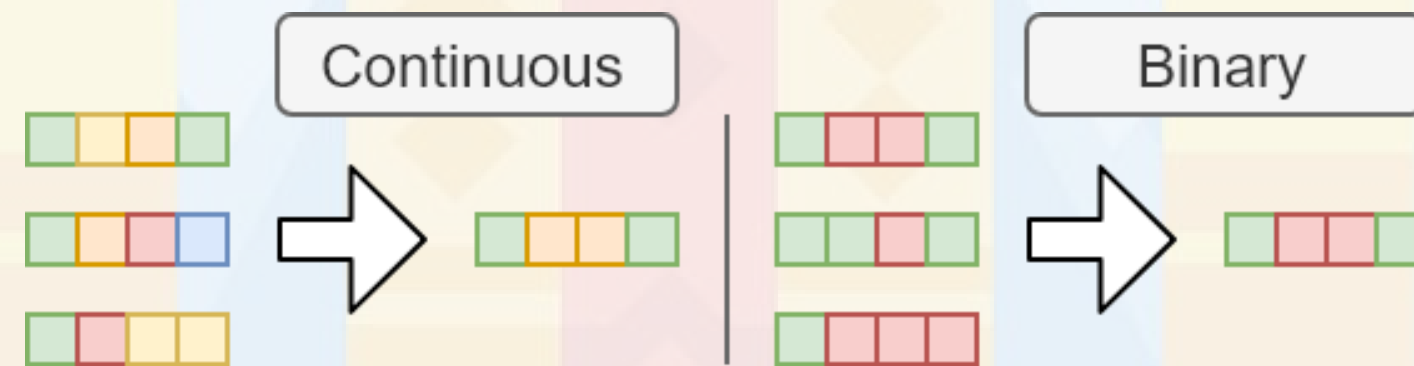
Addendum: Using R

- The same package, `xgboost` works for this in R
 - The level of support across R and python is the same
- Instead of using `sklearn` for hyperparameter tuning, you can use `caret` or `parsnip`

Ensembling

What are ensembles?

- Ensembles are models made out of models
- Ex.: You train 3 models using different techniques, and each seems to work well in certain cases and poorly in others
 - If you use the models in isolation, then any of them would do an OK (but not great) job
 - If you make a model using all three, you can get better performance if their strengths all shine through
- Ensembles range from simple to complex
 - Simple: a (weighted) average of a few model's predictions



When are ensembles useful?

1. You have multiple models that are all decent, but none are great
 - And, ideally, the models' predictions are not highly correlated

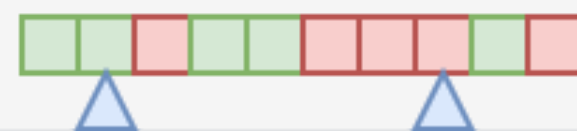
Suppose this is the vector to be predicted



Each of these is 60% accurate, and the average correlation between them is 17% (Errors are marked by blue triangles)



The most voted has 80% accuracy



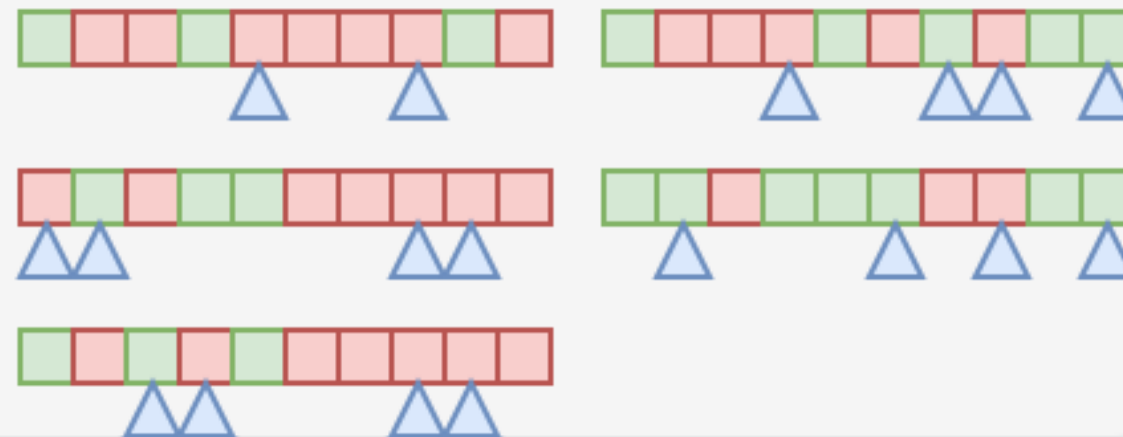
When are ensembles useful?

- 2. You have a really good model and a bunch of mediocre models
 - And, ideally the mediocre models are not highly correlated

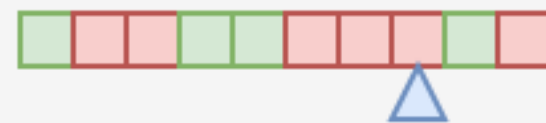
Suppose this is the vector to be predicted



The first model is 80% accurate, the others are 60% accurate and 32% correlated (Errors are marked by blue triangles)



Requiring unanimity to overpower the great model: 90%

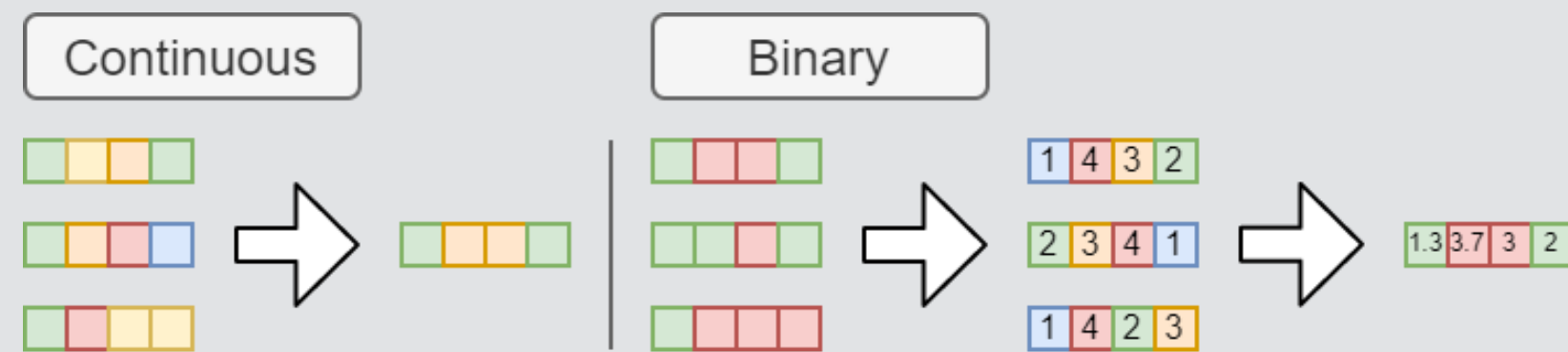


When are ensembles useful?

3. You really need to get just a bit more accuracy/less error out of the model, and you have some other models lying around
4. You want a more stable model
 - It helps to stabilize predictions by limiting the effect of errors or outliers produced by any one model on your prediction
 - Think: Diversification

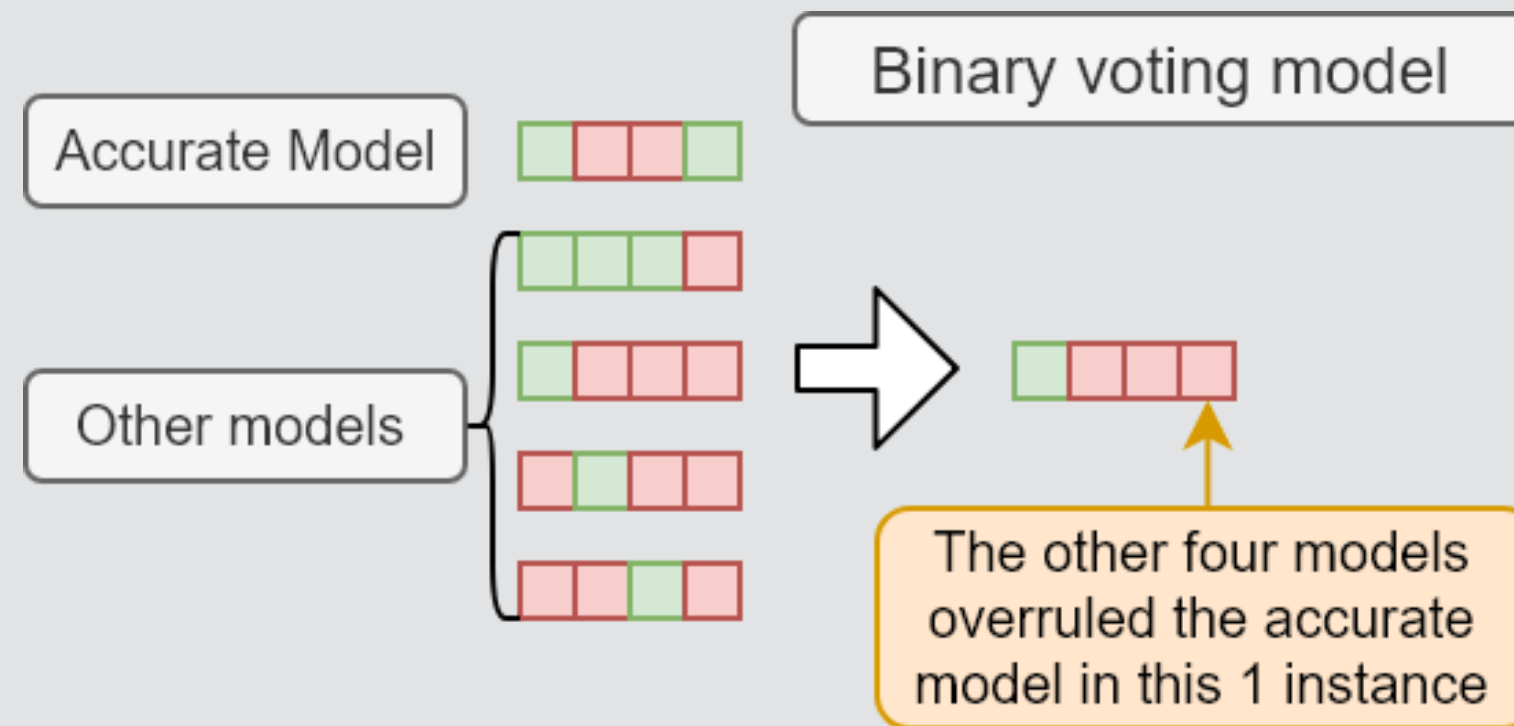
A simple ensemble (averaging)

- For continuous predictions, simple averaging is viable
 - Often you may want to weight the best model a bit higher
- For binary or categorical predictions, consider averaging *ranks*
 - i.e., instead of using a probability from a logit, use ranks 1, 2, 3, etc.
 - Ranks average a bit better, as scores on binary models (particularly when evaluated with measures like AUC) can have extremely different variances across models
 - In which case the ensemble is really just the most volatile model's prediction...
 - Not much of an ensemble



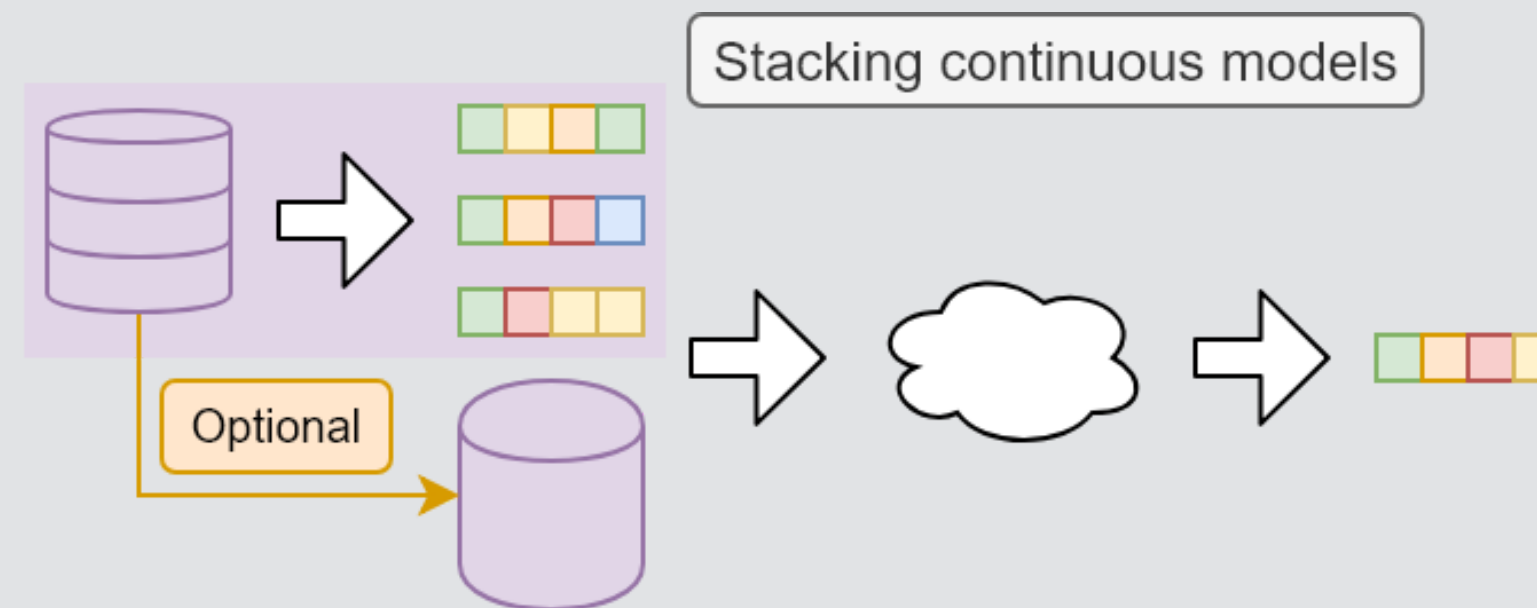
A more complex ensemble (voting model)

- If you have a model that is very good at predicting a binary outcome, ensembling can still help
 - This is particularly true when you have other models that capture different aspects of the problem
- Let the other models vote against the best model, and use their prediction if they are above some threshold of agreement



A lot more complex ensemble

- Stacking models (2 layers)
 1. Train models on subsets (folds) of the training data
 2. Make predictions for each model on the folds it wasn't applied to
 3. Train a new model that takes those predictions as inputs (and optionally the dataset as well)
- Blending (similar to stacking)
 - Like stacking, but using predictions on a hold out sample instead of folds (and thus all models are using the same data for predictions)



A simple averaging ensemble of our models

```
test_X_ens = pd.DataFrame({'XGBoost': final.predict_proba(test_X_logistic)[: ,1],
                          'LinearSVC': logistic(grid_svc.decision_function(test_X_logistic)),
                          'EN': reg_EN.predict_proba(test_X_logistic)[: ,1],
                          'lasso': reg_lasso.predict_proba(test_X_logistic)[: ,1],
                          'logistic': fit_logit.predict(test[vars_logistic])})

rank_X_ens = test_X_ens.rank()
arank_X_ens = rank_X_ens.XGBoost + rank_X_ens.LinearSVC + rank_X_ens.EN + rank_X_ens.lasso + rank_X_ens.logistic
auc = metrics.roc_auc_score(test_Y_logistic, arank_X_ens)
fpr, tpr, thresholds = metrics.roc_curve(test_Y_logistic, arank_X_ens)
display = metrics.RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=auc)
display.plot()
```



Practicalities

- Methods like stacking or blending are much more complex than a simple averaging or voting based ensemble
 - But in practice they perform slightly better

Recall the tradeoff between complexity and accuracy!

- As such, we may not prefer the complex ensemble in practice, unless we only care about accuracy

Example: In 2009, Netflix awarded a \$1M prize to the BellKor's Pragmatic Chaos team for beating Netflix's own user preference algorithm by >10%. The algorithm was so complex that Netflix **never used it**. It instead used a simpler algorithm with an 8% improvement.

[Geoff Hinton's] Dark knowledge

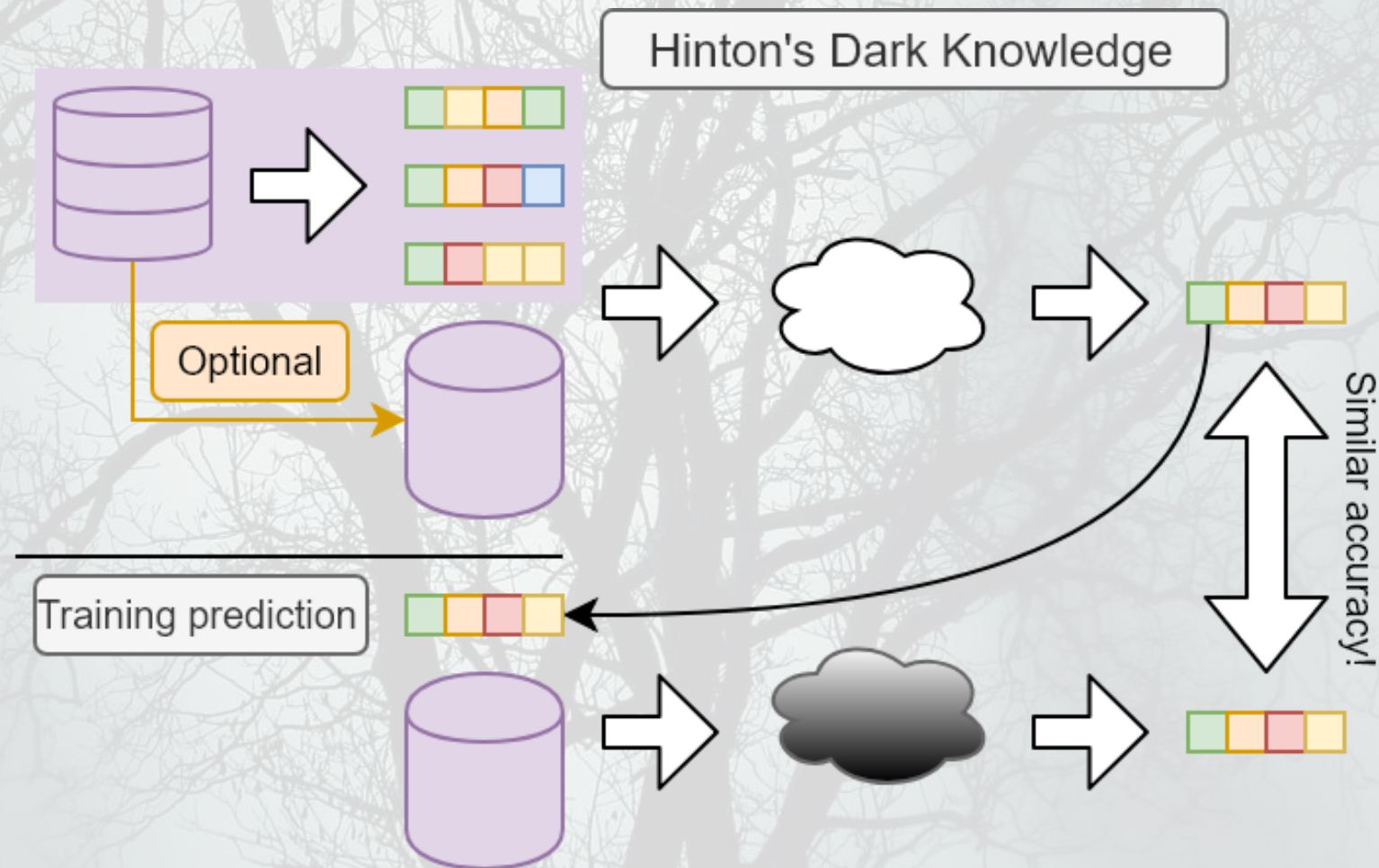
- Complex ensembles work well
- Complex ensembles are exceedingly computationally intensive
 - This is bad for running on small or constrained devices (like phones)

Dark knowledge

- We can (almost) always create a simple model that approximates the complex model
 - Interpret the above literally – we can train a model to fit the model

Dark knowledge

- Train the simple model not on the actual DV from the training data, but on the best algorithm's (softened) prediction for the training data
- Somewhat surprisingly, this new, simple algorithm can work almost as well as the full thing!



An example of this dark knowledge

- Google's full model for interpreting human speech is >100GB
 - As of October 2019
- In Google's Pixel 4 phone, they have human speech interpretation running locally *on the phone*
 - Not in the cloud like it works on any other Android phone

How did they do this?

- They can approximate the output of the complex speech model using a 0.5GB model
- 0.5GB isn't small, but it's small enough to run on a phone

Learning more about Ensembling

- [Scikit-learn's documentation on ensemble methods it supports](#)
- [Geoff Hinton's Dark Knowledge slides](#)
 - For more details on *dark knowledge*, applications, and the softening transform
 - His interesting (though highly technical) [Reddit AMA](#)
- [A Kagglers Guide to Model Stacking in Practice](#)
 - A short guide on stacking with nice visualizations
- [Kaggle Ensembling Guide](#)
 - A comprehensive list of ensembling methods with some code samples and applications discussed
- [Ensemble Learning to Improve Machine Learning Results](#)
 - Nicely covers bagging and boosting (two other techniques)

There are many ways to ensemble, and there is no specific guide as to what is best. It may prove useful in the group project, however.

Addendum: Using R

- There are a couple interesting packages in R for ensembling:
 - The `Superlearner` package aims to automate building ensembles
 - Think of it like an automated cross-validation for ensemble construction
 - The `EnsembleML` package allows you to specify an ensemble and train the underlying models together
- You can also roll your own ensemble as we did in the example earlier

Conclusion



Exercise

1. Import a file from one of your papers using Pandas:
 - csv file using `pd.read_csv()`
 - Stata using `pd.read_stata()`
 - SAS using `pd.read_sas()`
2. Pick a regression to replicate
 - Ideally one without fixed effects (or where they are not critical)
 - Or one hot encode the fixed effects using `pd.get_dummies()`
3. Define a variable that is a list of every IV and control in the regression
4. Replicate some or all of the following analyses, using the code from the jupyter notebooks
 - Logistic regression, LASSO, elastic net, SVM (SVC or SVR), XGBoost
 - If you are replicating a linear model, replace `reg:logistic` with `reg:linear`

Wrap-up

SVM

- A flexible model for classification
- Good when our interest is in getting the most observations correctly classified

XGBoost

- An extremely flexible and efficient non-linear algorithm
- Very capable for classification or in-sample regression
- Able to solve a wide variety of problems

Ensembling

- Combining algorithms to produce a more stable (and sometimes more accurate) model

Packages used for these slides

Python

- matplotlib
- numpy
- pandas
- scikit-learn
- statsmodels
- umap-learn
- xgboost

R

- kableExtra
- knitr
- reticulate
- revealjs

References

- Bao, Yang, and Anindya Datta. “Simultaneously discovering and quantifying risk types from textual risk disclosures.” *Management Science* 60, no. 6 (2014): 1371-1391.
- Brown, Nerissa C., Richard M. Crowley, and W. Brooke Elliott. “What are you saying? Using topic to detect financial misreporting.” *Journal of Accounting Research* 58, no. 1 (2020): 237-291.
- Chernozhukov, Victor, Mert Demirer, Esther Duflo, and Ivan Fernandez-Val. Generic machine learning inference on heterogenous treatment effects in randomized experiments. No. w24678. National Bureau of Economic Research, 2018
- Deryugina, Tatyana, Garth Heutel, Nolan H. Miller, David Molitor, and Julian Reif. “The mortality and medical costs of air pollution: Evidence from changes in wind direction.” *American Economic Review* 109, no. 12 (2019): 4178-4219.
- Einav, Liran, Amy Finkelstein, Sendhil Mullainathan, and Ziad Obermeyer. “Predictive modeling of US health care spending in late life.” *Science* 360, no. 6396 (2018): 1462-1465.
- McInnes, Leland, John Healy, and James Melville. “Umap: Uniform manifold approximation and projection for dimension reduction.” arXiv preprint arXiv:1802.03426 (2018).

Custom code

```
# Side-by-side plot of UMAP coloring by predictions and actual values

# Cut down version of umap.plot.points to remove dependencies on datashader, bokeh, holoviews, scikit-image, and colorcet
def umap_compare_svm(X, Yhat, Y, clip = None, cmap='viridis', subset=None, binary=False, title=None):
    reducer = umap.UMAP()
    umap_object = reducer.fit(X)
    embed = _get_embedding(umap_object)
    if clip is not None:
        Yhat = np.clip(Yhat, clip[0][0], clip[0][1])
        Y = np.clip(Y, clip[1][0], clip[1][1])

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8,4))
    if subset is not None:
        embed_X = embed[subset,0]
        embed_Y = embed[subset,1]
        Y = np.array(Y[subset])
        X = np.array(X[subset])
        Yhat = np.array(Yhat[subset])
    else:
        embed_X = embed[:, 0]
        embed_Y = embed[:, 1]

    point_size = 100.0 / np.sqrt(len(embed_X))

    if binary:
        point_size = point_size * (1 + Y * binary)

    # color by values for Yhat
    points = ax1.scatter(embed_X, embed_Y, s=point_size, c=Yhat, cmap=cmap)
    fig.colorbar(points, ax=ax1, orientation='horizontal')

    ax1.set(xticks=[], yticks=[])
    ax1.set_title("Predicted values")

    # color by values for Y
    points = ax2.scatter(embed_X, embed_Y, s=point_size, c=Y, cmap=cmap)

    fig.colorbar(points, ax=ax2, orientation='horizontal')

    ax2.set(xticks=[], yticks=[])
    ax2.set_title("Actual values")

    if title is not None:
        fig.suptitle(title)

    if clip is not None:
        foot = 'Predicted values winsorized to [{}], {}; Actual values winsorized to [{}], {}'.format(clip[0][0], clip[0][1], clip[1][0], clip[1][1])
        plt.figtext(0.2, 0.3, foot, horizontalalignment='left')

    return (ax1, ax2)
```



Custom code

```
# Replication of R's coefplot function for use with sklearn's linear and logistic LASSO
def coefplot(names, coef, title=None):
    # Make sure coef is list, cast to list if needed.
    if isinstance(coef, np.ndarray):
        if len(coef.shape) > 1:
            coef = list(coef[0])
        else:
            coef = list(coef)

    # Drop unneeded vars
    data = []
    for i in range(0, len(coef)):
        if coef[i] != 0:
            data.append([names[i], coef[i]])
    data.sort(key=lambda x: x[1])

    # Add in a key for the plot axis
    data = [data[i] + [i+1] for i in range(0, len(data))]

    fig, ax = plt.subplots(figsize=(6, 0.25*len(data)+0.25), constrained_layout=True)

    ax.scatter([i[1] for i in data], [i[2] for i in data])

    ax.grid(axis='y')
    ax.set(xlabel="Fitted value", ylabel="Residual", title=(title if title is not None else "Coefficient Plot"))

    ax.axvline(x=0, linestyle='dotted')
    ax.set_yticks([i[2] for i in data])
    ax.set_yticklabels([i[0] for i in data])

    return ax
```



Custom code

```
# Helper functions

# Logistic function in numpy
def logistic(x):
    return 1 / (1 + np.exp(-1 * x))

# From umap.plot source code on Github
def _get_embedding(umap_object):
    if hasattr(umap_object, "embedding_"):
        return umap_object.embedding_
    elif hasattr(umap_object, "embedding"):
        return umap_object.embedding
    else:
        raise ValueError("Could not find embedding attribute of umap_object")
```

