ML for SS: Classification

Dr. Richard M. Crowley
rcrowley@smu.edu.sg
https://rmc.link/

Overview

Papers

> Paper 1: Purda and Skillicorn 2015

- A fairly approachable overview of ML methods in economics
- The points the paper makes are applicable broadly in any archival/empirical discipline

> Paper 2: Noh et al. 2021

- Uses XGBoost for an interesting application: Determining fall risk in older adults
- Combines experimental work with ML-based econometrics

Technical Discussion: Classification

- SVM and Tree-based algorithms (like XGBoost)

Python

- Using `sklearn` for SVM
- Using `xgboost` for XGBoost
- Using `sklearn` for hyperparameter tuning

R

- Using caret for SVM
- Using xgboost for XGBoost
- Using tidymodels and related packages for hyperparameter tuning

> Python is generally a bit stronger for these topics.

> There is a fully worked out solution for each language on my website, data is on eLearn.

Main application: Binary problem

- 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)

> Same problem and data as last week's binary problem

Main application: A Linear problem

- 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)

Same problem and data as last week’s linear problem

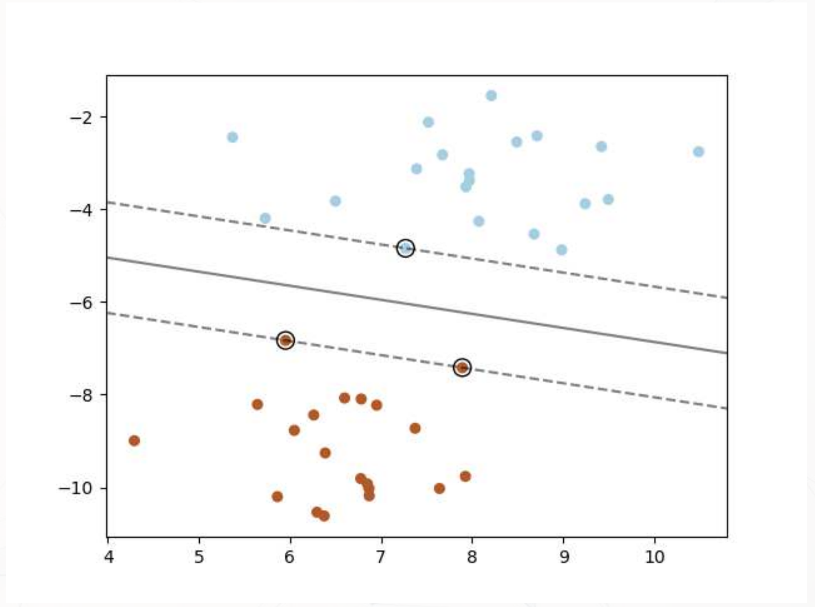
SVM: Support Vector Machine

What is SVM?

> Simpler case: Binary Classification

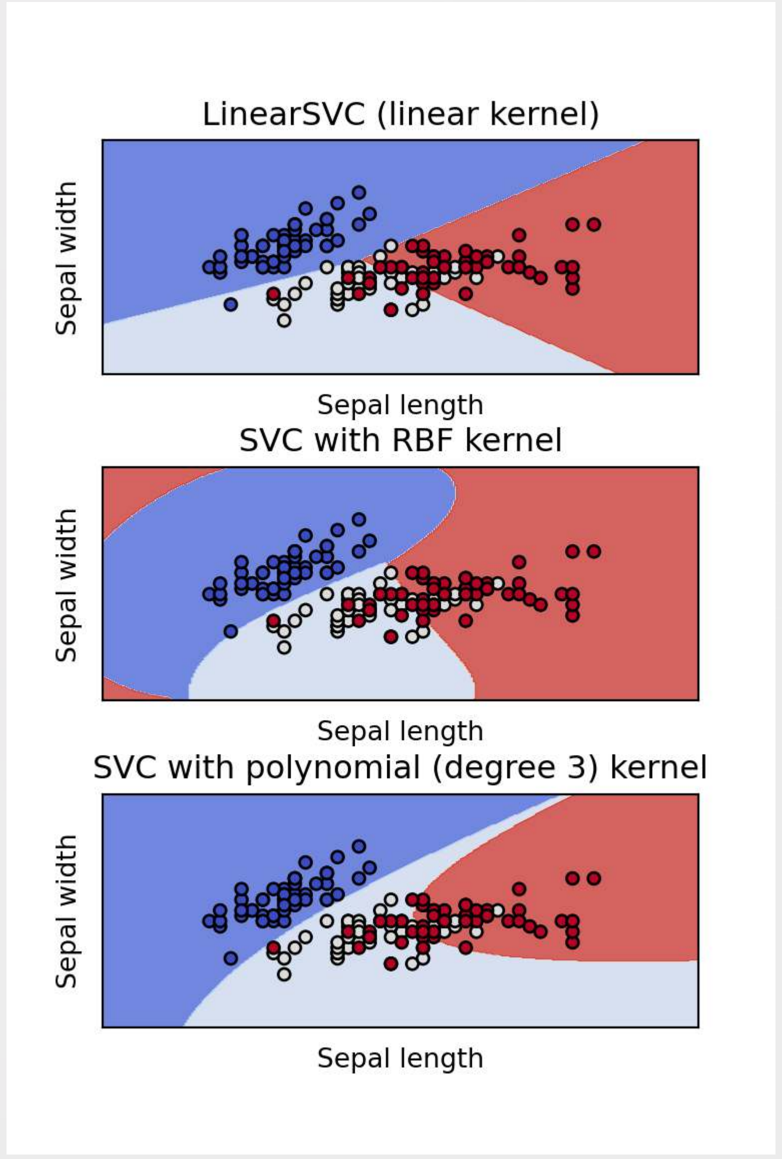
- SVM-type algorithms generally focus on separability under some tolerance for error
  - This is quite different from our regression approaches
    - Regression focuses on *minimizing an error function*

- 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



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
    - More than 10,000 observations leads to a lot of slow down for non-linear kernels
3. Difficult to interpret model when using a non-linear kernel
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; time increase for additional obs. is polynomial
- The results aren't quite the same across backends
  - `liblinear` uses a penalized intercept while `libsvm` does not
  - `liblinear` optimizes a "squared hinge" loss function while `libsvm` optimizes "hinge" loss

> 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

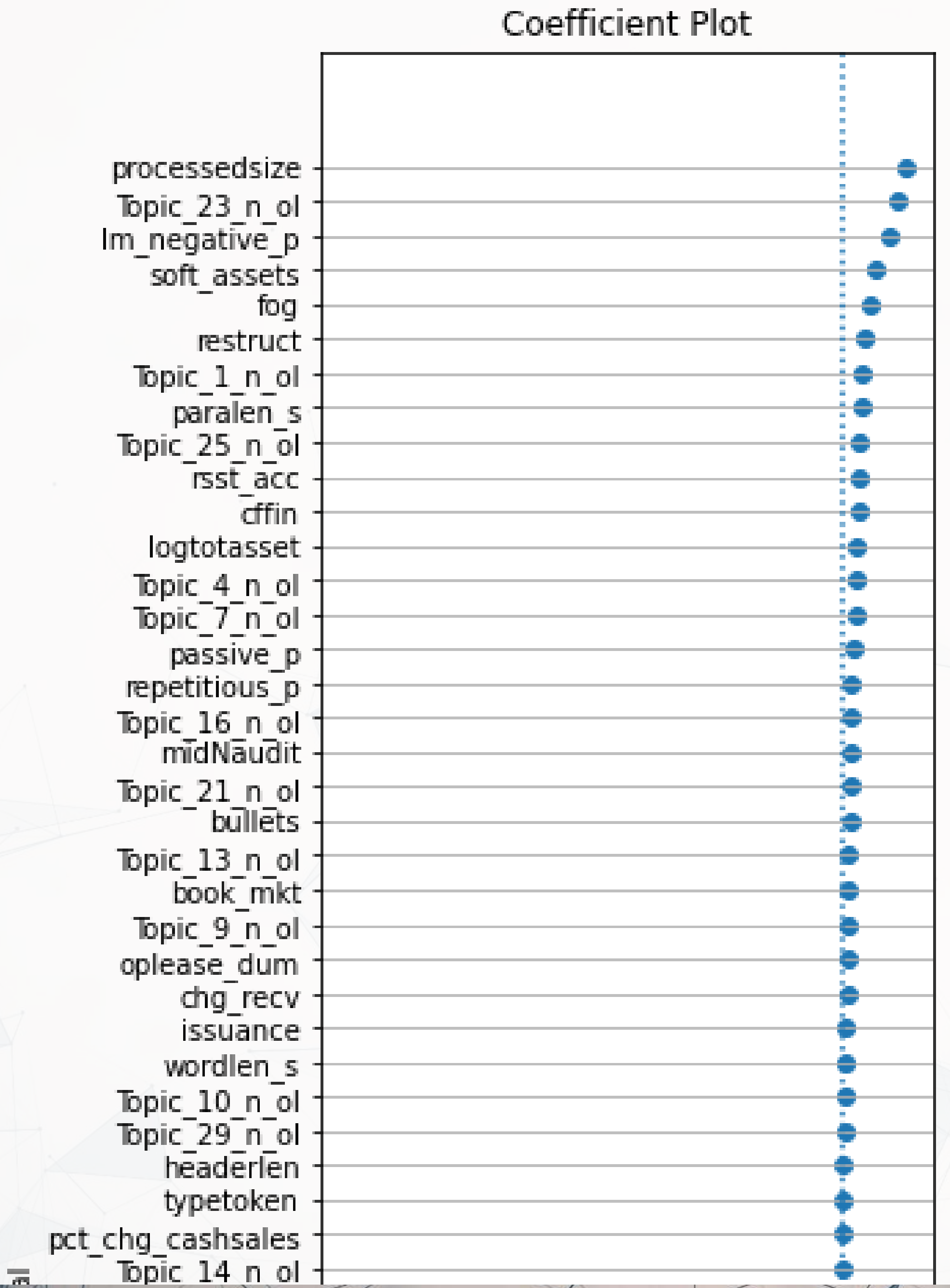
```python
model_svc = svm.LinearSVC(C=1, dual=False)
model_svc.fit(train_X_logistic, train_Y_logistic)
```

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

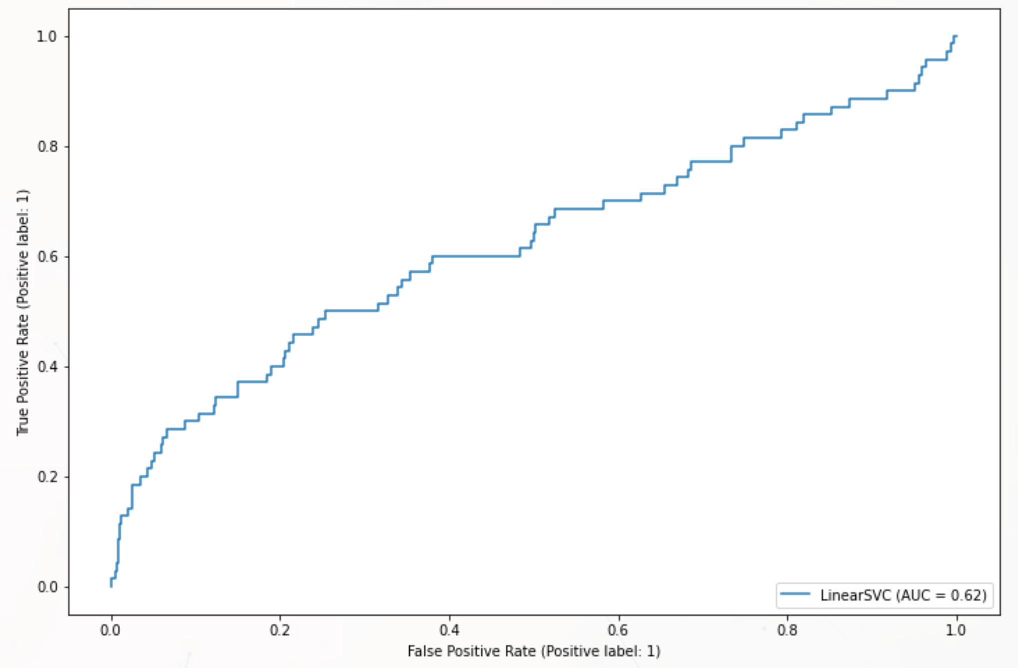
```python
coefplot(vars_logistic, model_svc.coef_)
```

Visualizing LinearSVC for irregularity detection

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



```
display = \
  metrics.RocCurveDisplay.from_estimator(
  model_svc, test_X_logistic,
  test_Y_logistic)
display.plot()
```



Optimizing the C parameter

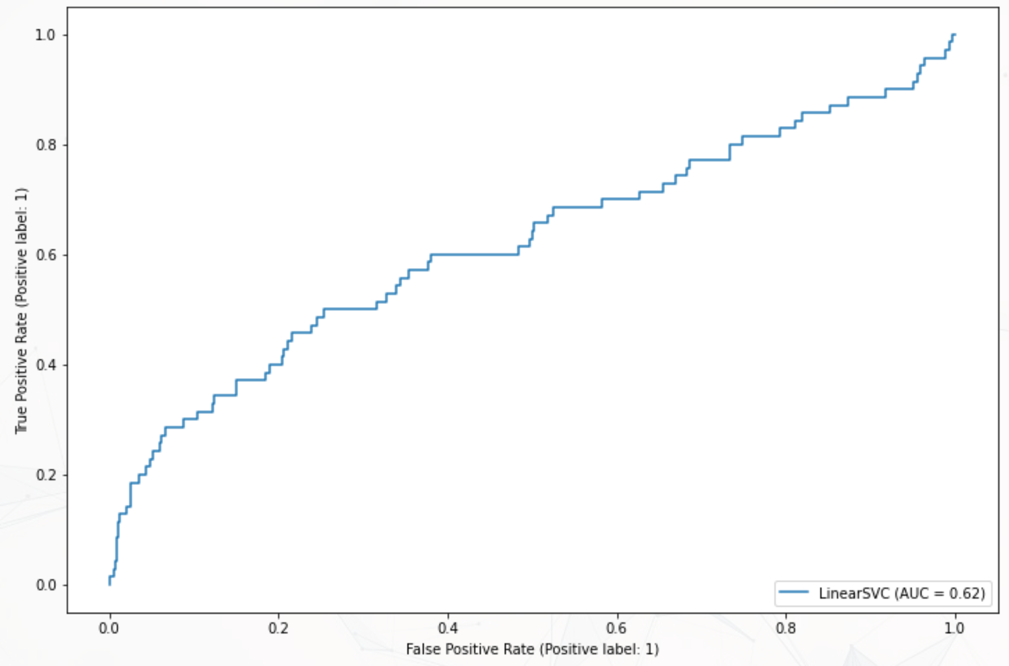
```python
C_range = np.logspace(-2, 6, 9)
param_grid = dict(C=C_range)
cv = model_selection.StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=1)
grid_svc = model_selection.GridSearchCV(svm.LinearSVC(dual=False), param_grid=param_grid, cv=cv)
grid_svc.fit(train_X_logistic, train_Y_logistic)
print("The best parameter is C=%s with a score of %0.2f"
      % (grid_svc.best_params_['C'], grid_svc.best_score_))
```

```
[1] "The best parameter is C=0.01 with a score of 0.99"
```

Comparison pre- vs post-optimization: ROC

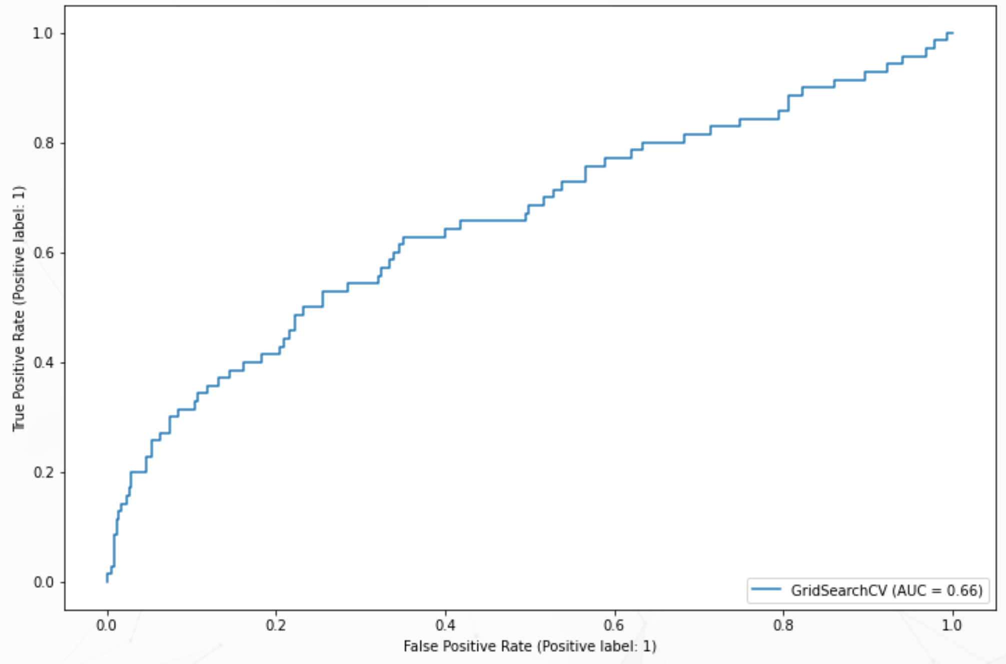
Unoptimized

```
display = \
  metrics.RocCurveDisplay.from_estimator(
  model_svc, test_X_logistic,
  test_Y_logistic)
display.plot()
```



Optimized

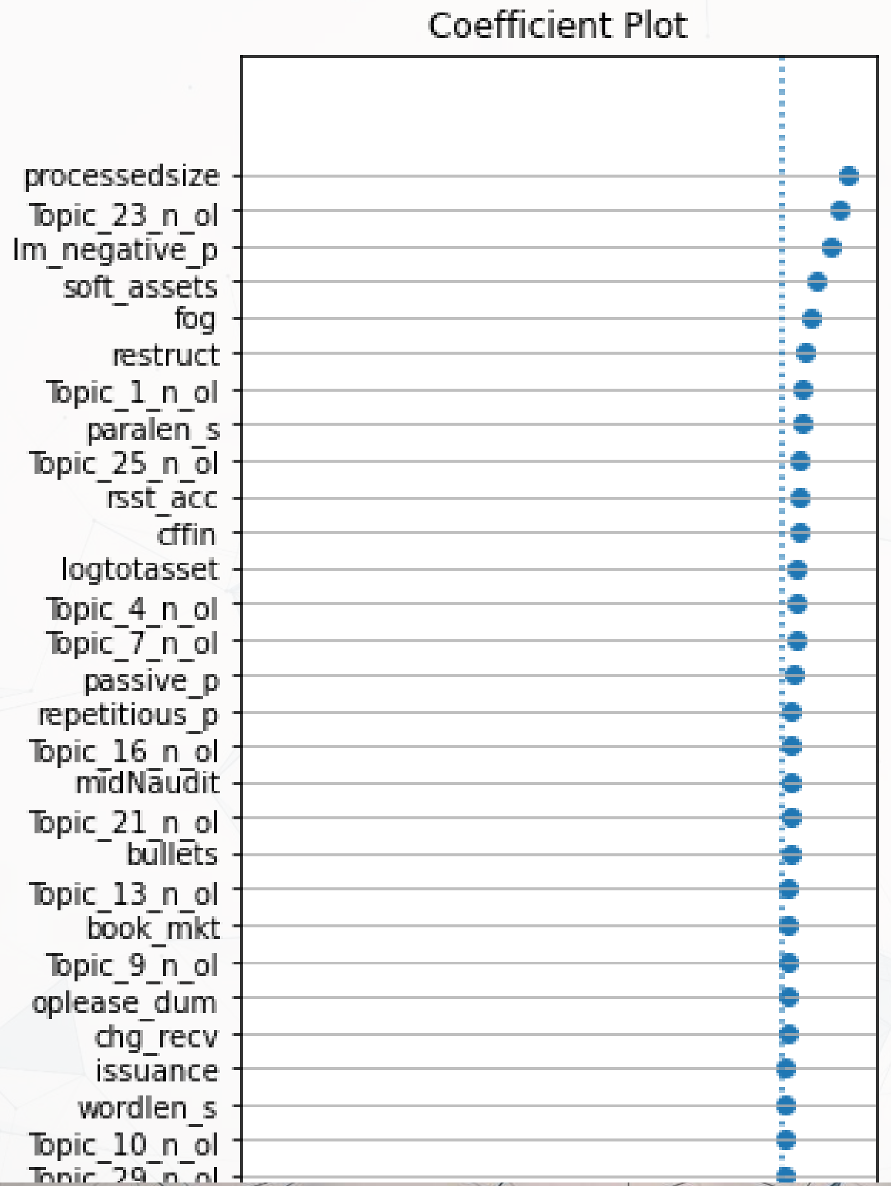
```
display = \
  metrics.RocCurveDisplay.from_estimator(
  grid_svc, test_X_logistic,
  test_Y_logistic)
```



Comparison pre- vs post-optimization: Coefficients

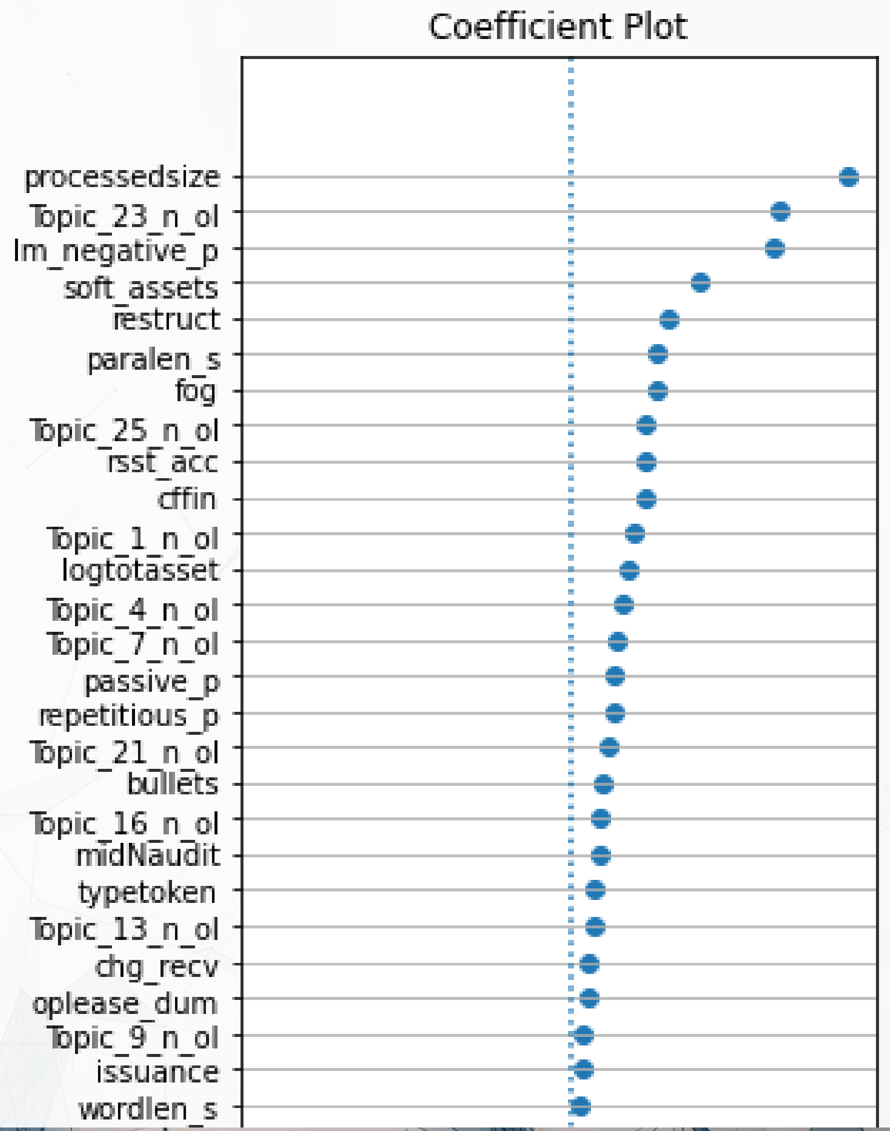
Unoptimized

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



Optimized

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



Visualizing with UMAP

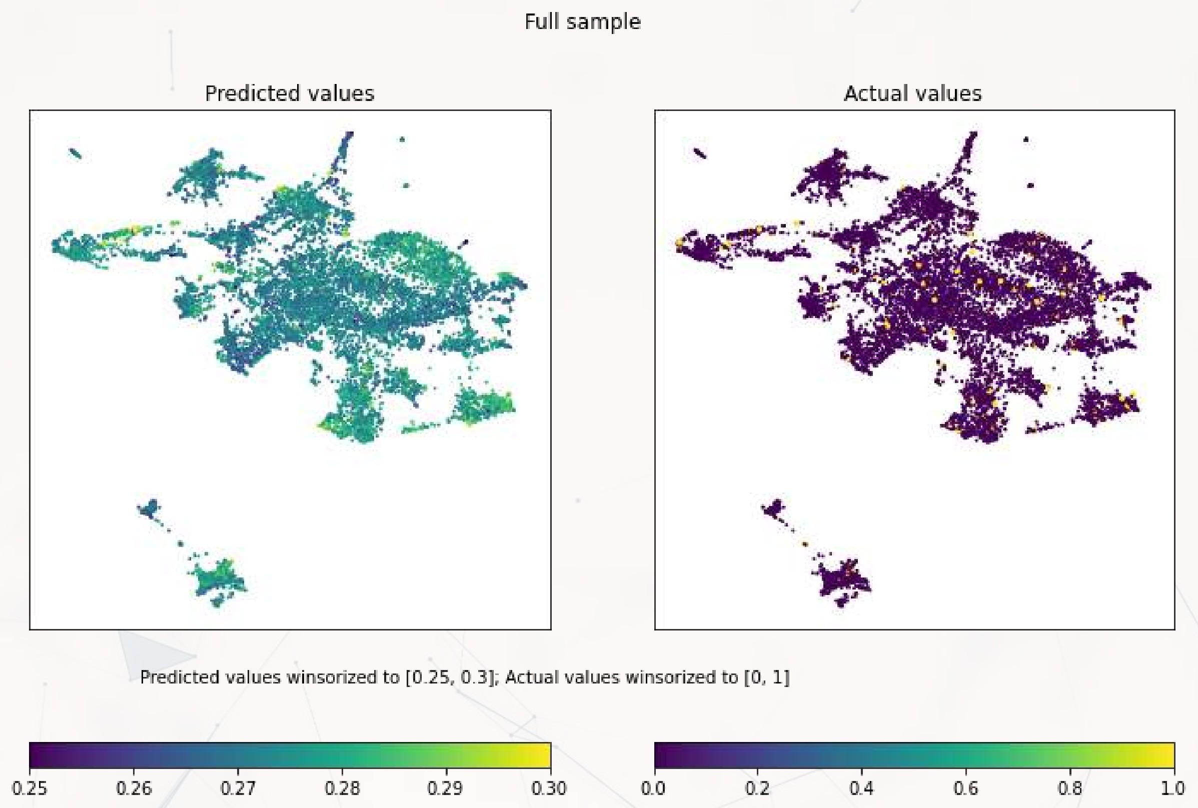
> What is UMAP?

- UMAP stands for Uniform Manifold Approximation and Projection for Dimension Reduction
  - From Leland, Healy and Melville (2018) (8k+ 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 Reimannian manifolds and tries to maintain geodesic distance around a point – it is well supported theoretically

Visualizing what SVM is doing using UMAP

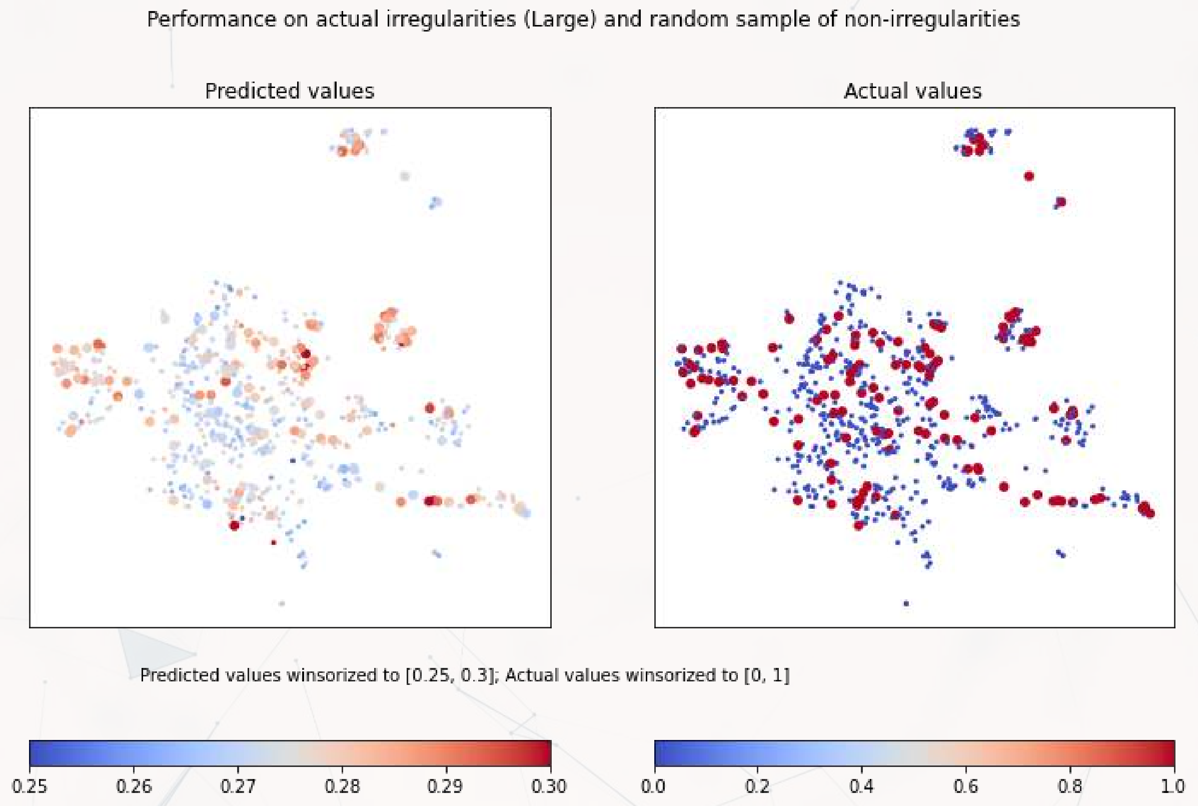
```python
train_Yhat_logistic = logistic(grid_svc.decision_function(train_X_logistic))
umap_compare_svm(train_X_logistic, train_Yhat_logistic, train_Y_logistic,
                 clip=[[0.25, 0.3], [0, 1]], binary=5, title="Full sample")
```



The data is really noisy

Visualizing what SVM is doing using UMAP

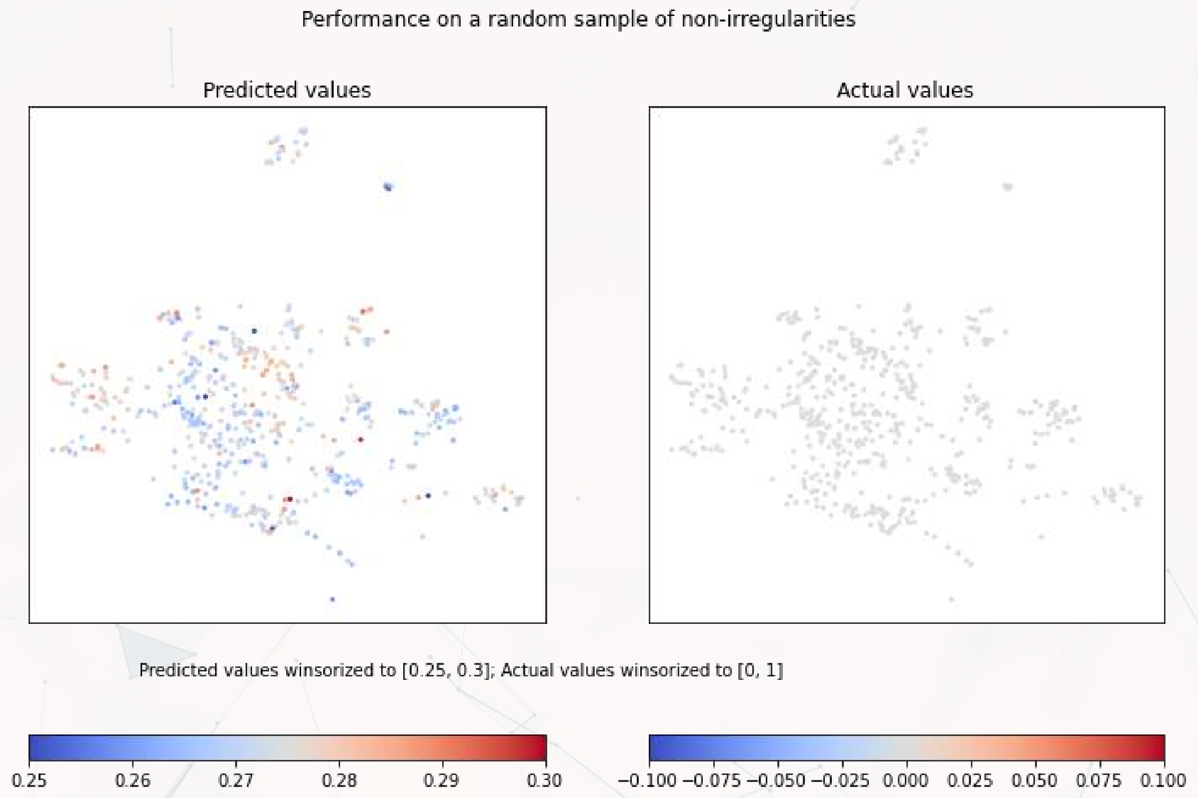
```
umap_compare_svm(train_X_logistic, train_Yhat_logistic, train_Y_logistic, clip=[[0.25, 0.3], [0, 1]],
  cmap='coolwarm', binary=5, subset=((train_Y_logistic==1) | (np.random.rand(len(train_Y_logistic))<0.05)),
  title="Performance on actual irregularities (Large) and random sample of non-irregularities")
```



> 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=
                 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

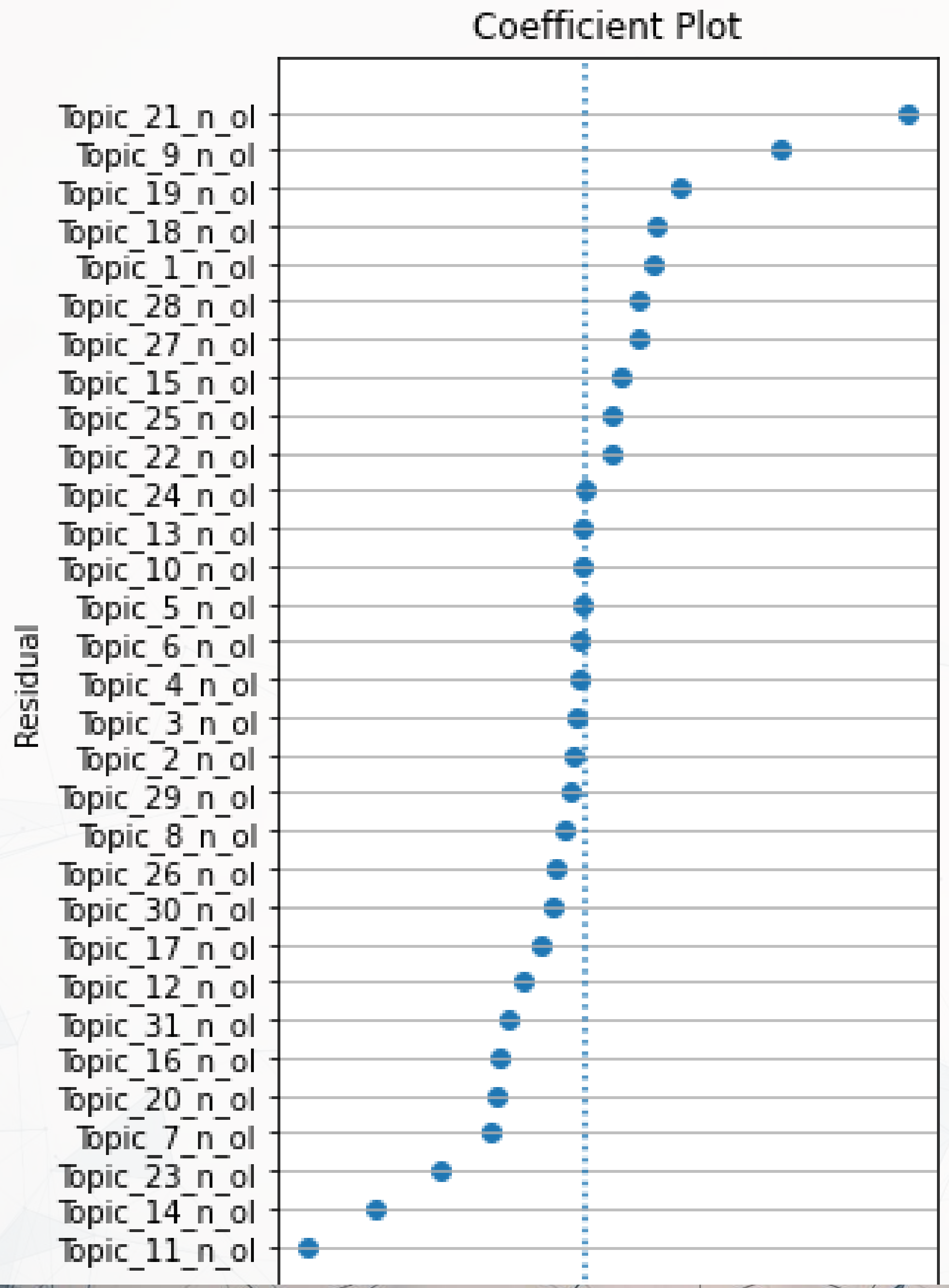
```python
model_svr = svm.LinearSVR(C=1, dual=False,
  loss='squared_epsilon_insensitive')
model_svr.fit(train_X_linear,
              np.ravel(train_Y_linear))
```

```python
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))
print("The best parameter is C=%s " +
      "with a score of %0.2f" % (
        grid_svr.best_params_['C'],
        grid_svr.best_score_))
```

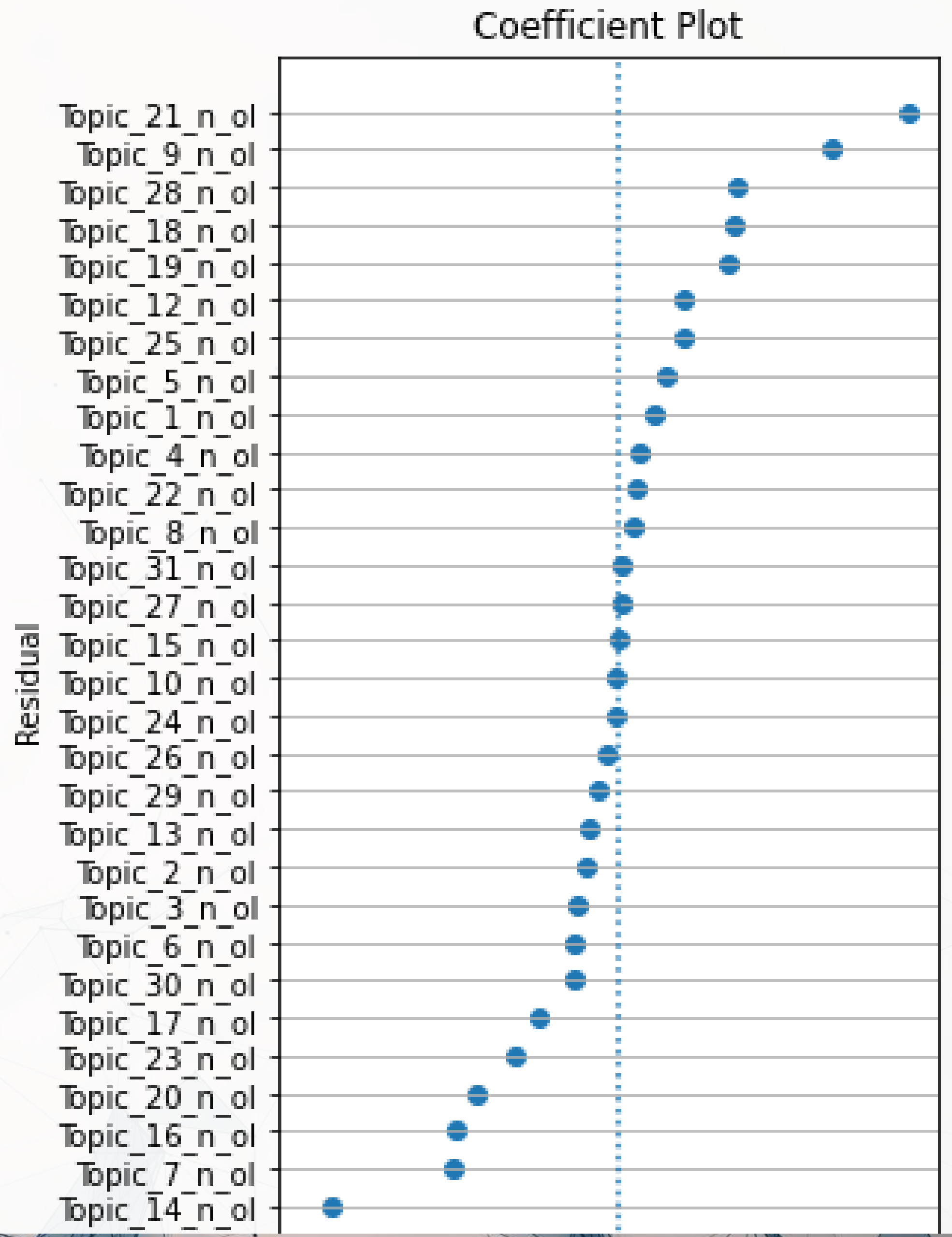
```
[1] "The best parameter is C=0.0001 with a score
of 0.06"
```

SVR coefficients

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

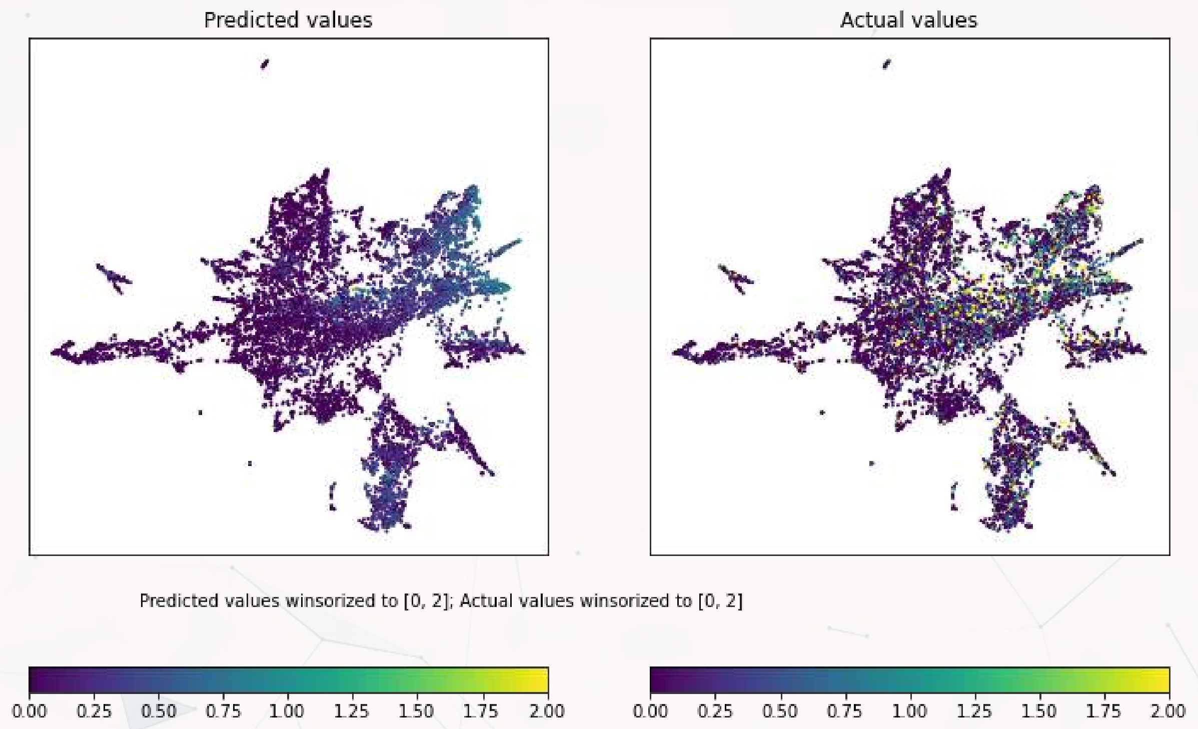


```python
coefplot(vars_linear,
         grid_svr.best_estimator_.coef_)
```



Visualizing SVR with UMAP

```python
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

Using R for the above

- We can use tidymodels to handle training of the model
  - It will offload the model computation to `kernlab`
- tidymodels is a collection of packages intended to serve as a spiritual successor to caret
- It is a collection of packages aimed at making ML workflows easier in R, much like what Scikit-learn does for python
  - `parsnip`, `recipes`, `rsample`, `dials`, `yardstick`, etc.
- It is still rough around the edges, but it is fairly functional

Step 1: Make a recipe for your data

- *Recipes* serve as a guide on how to preprocess your data
  - There are many possible steps
- This keeps preprocessing quick and transparent

```r
recipe_svm <-
  recipe(BCE_eq, data = train)  %>%
  step_zv(all_predictors()) %>% # remove any zero variance predictors
  step_center(all_predictors()) %>%  # Center all prediction variables
  step_scale(all_predictors()) %>%  # Scale all prediction variables
  step_intercept() %>% # Add an intercept to the model
  step_num2factor(all_outcomes(), ordered = T, levels=c("0","1"),
                  transform = function(x) x + 1, skip = TRUE)  # Convert DV to factor
```

Step 2: Define your ML model

- There are many built-in models in tidymodels
- For SVM, we will use `svm_linear`
  - Note how we specify `tune()` to the `cost` parameter
    - This is how we tell it where the grid search will go later!
- Setting mode to classification ensures we use something like SVC rather than SVR
- We can change the backend package by setting a different engine, with minimal changes needed to the rest of our code!

```r
model_svm <-
  svm_linear(cost = tune()) %>%
  set_mode("classification") %>%
  set_engine("kernlab")
```

Step 3: Define a workflow

- Workflows piece together the larger elements of a tidy model
- Simplifies some of the hassle of using functions across `tidymodels` packages

```r
workflow_svm <- workflow() %>%
  add_model(model_svm) %>%
  add_recipe(recipe_svm)
```

Step 4: Tie up loose ends

- We need to set a cross validation: `vfold_cv()`
- We need to specify the metric to track: `metric_set()`
- We need to set our grid search's grid: `expand_grid()`

```r
folds_svm <- vfold_cv(train, v=10)  # from rsample
metrics_svm = metric_set(roc_auc)  # from yardstick
grid_svm <- expand_grid(cost = exp(seq(-10,0, length.out=10)))
```

Step 5: Run the model

> We have everything we need to run the model

```r
svm_fit_tuned <- tune_grid(workflow_svm,
                           grid = grid_svm,
                           resamples = folds_svm,
                           metrics=metrics_svm)
```

- `tune_grid()` will execute the workflow:
  1. Standardize our training data
  2. Run the model
  3. Apply 10-fold CV to it
  4. Track ROC AUC for each model run
- The resulting fitted model can then be analyzed

See which model was the best

```r
show_best(svm_fit_tuned, metric = "roc_auc")
```

```
# A tibble: 5 × 7
       cost .metric .estimator  mean     n std_err .config
      <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1 0.000419  roc_auc binary     0.637    10  0.0259 Preprocessor1_Model03
2 0.000138  roc_auc binary     0.616    10  0.0266 Preprocessor1_Model02
3 0.0000454 roc_auc binary     0.606    10  0.0320 Preprocessor1_Model01
4 0.00387   roc_auc binary     0.605    10  0.0240 Preprocessor1_Model05
5 0.0117    roc_auc binary     0.599    10  0.0257 Preprocessor1_Model06
```

Step 6: Re-run the model with the full data

```r
svm_final <- workflow_svm %>%
  finalize_workflow(
  select_best(svm_fit_tuned, "roc_auc")
) %>%
  fit(train)
```

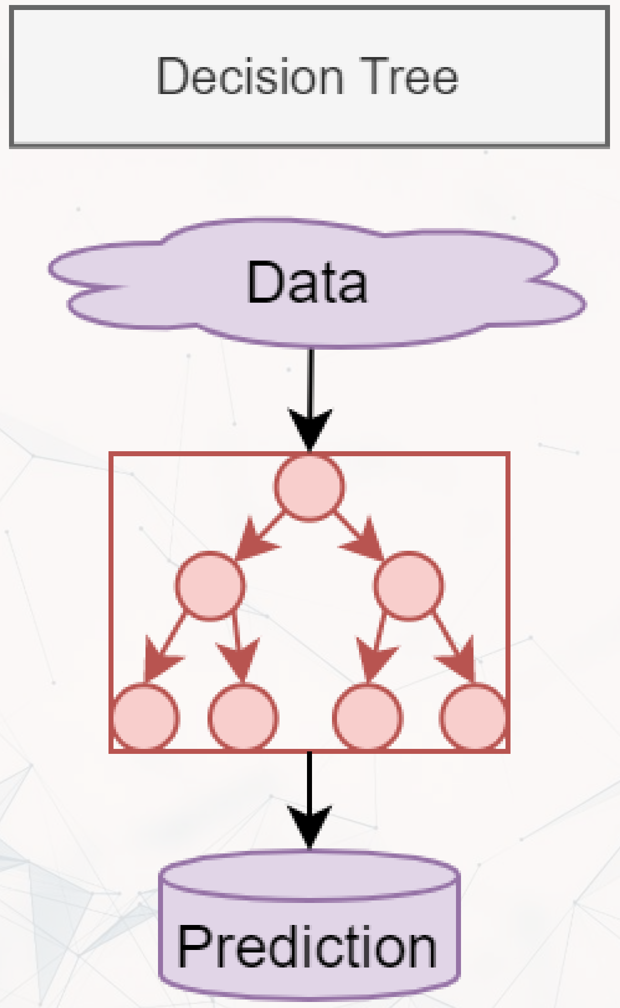
> You need to do this in order to be able to predict with the model

- The `svm_final` object can be used with the standard `predict()` function
  - The `svm_fit_tuned` object could not!

Tree-based models

Simplest model: Decision tree

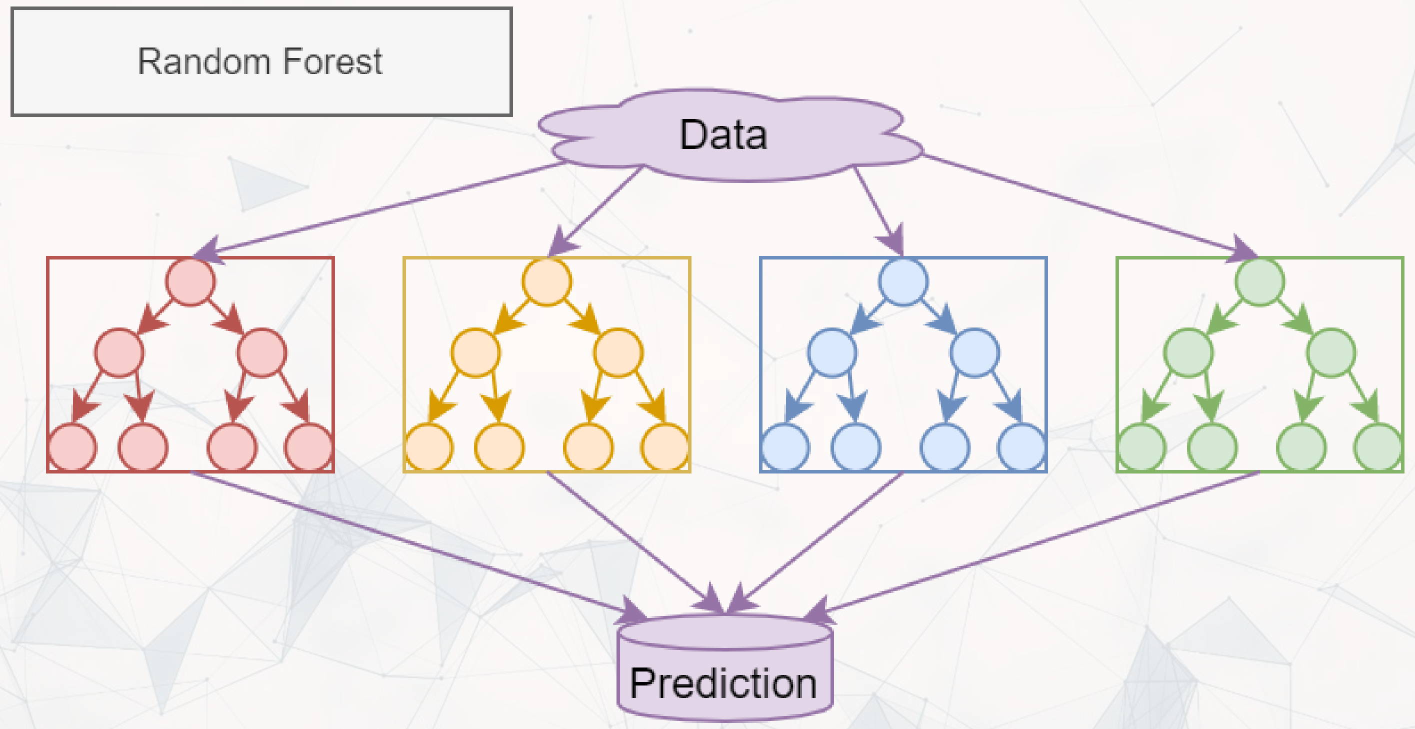
- A simple decision tree behaves as we saw in Mullainathan and Spiess (2017 JEP)
- It provides a set of conditions to traverse to go from data to the estimated output
- In order to capture a complex problem, many layers are needed



Simple model: Random Forest

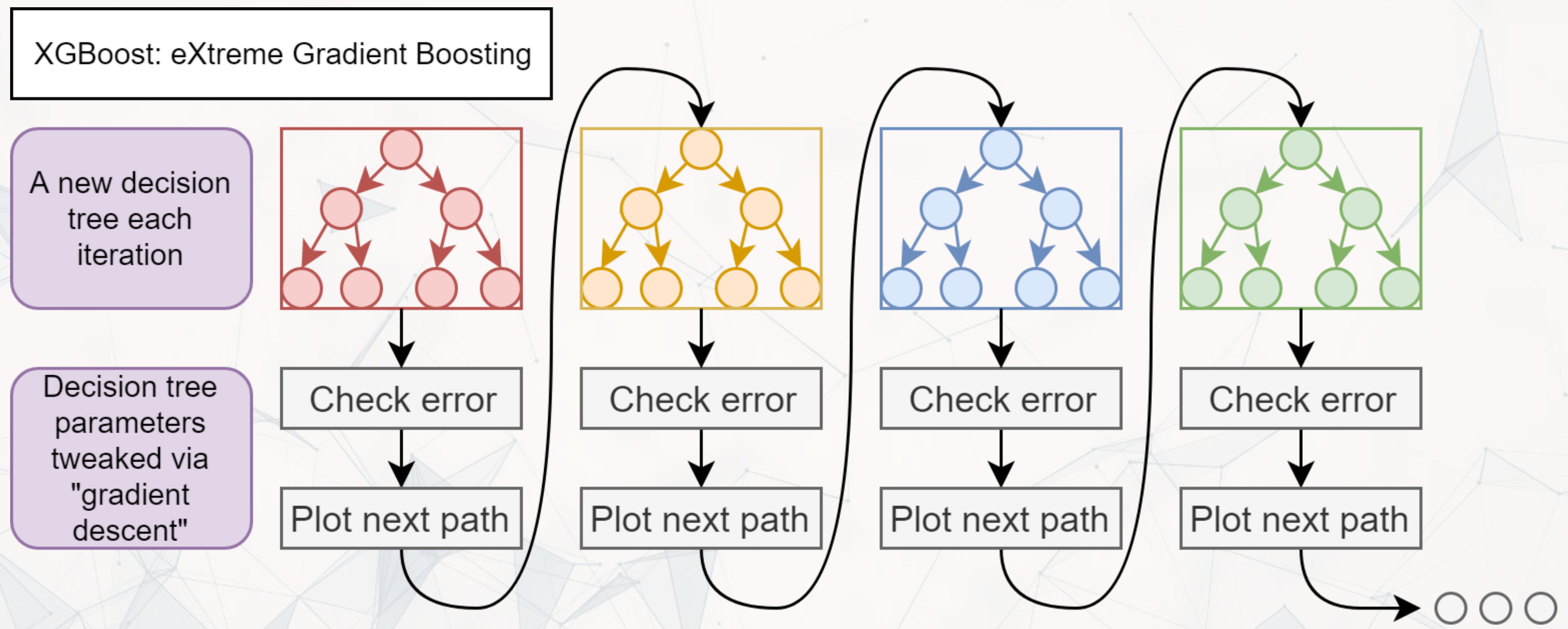
- 1 decision tree is OK, but…
  - There is a lot of error unless the tree is complex
  - Successive iterations of trees can be very different from one another

> Run a bunch of decision trees with less depth each and average them (but don’t give them all exactly the same data)



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: eXtreme Gradient Boosting
A new decision tree each iteration
Decision tree parameters tweaked via "gradient descent"
Check error
Plot next path
Check error
Plot next path
Check error
Plot next path
Check error
Plot next path

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 (R implementation only)
- 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 parameters

```python
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
```

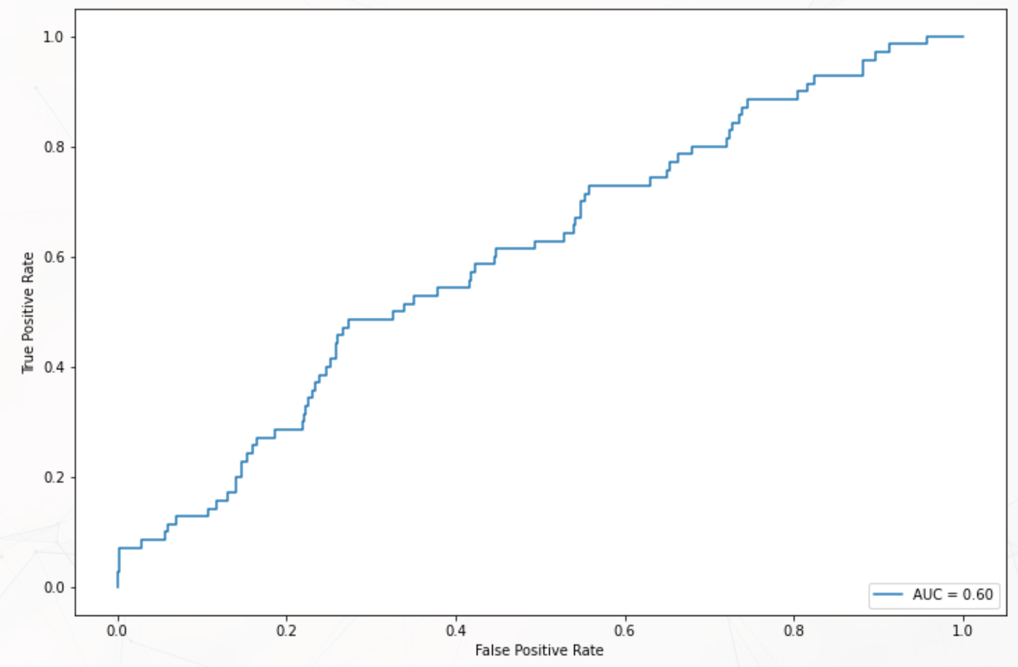
A lot of parameters – we can optimize all from `eta` to `colsample_bytree` and the number of rounds

Running XGBoost

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

```
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)

model_xgb_logistic = xgb.train(param, dtrain, num_round)
test_Yhat_xgb_logistic = model_xgb_logistic.predict(dtest)
fpr, tpr, thresholds = metrics.roc_curve(test_Y_logistic, test_Yhat_xgb_logistic)
display = metrics.RocCurveDisplay(fpr=fpr, tpr=tpr, roc_auc=auc)
```

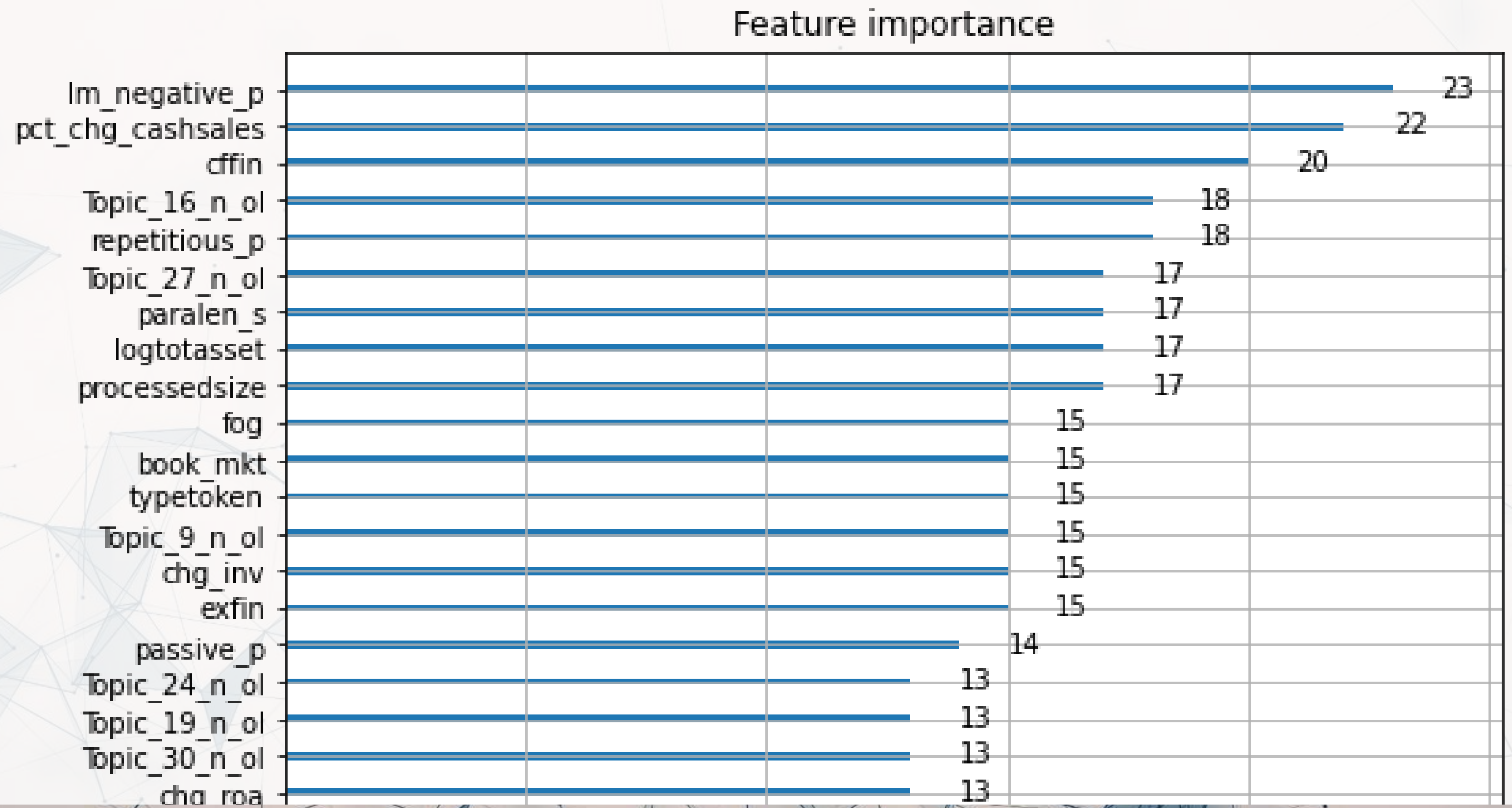
```
display.plot()
```



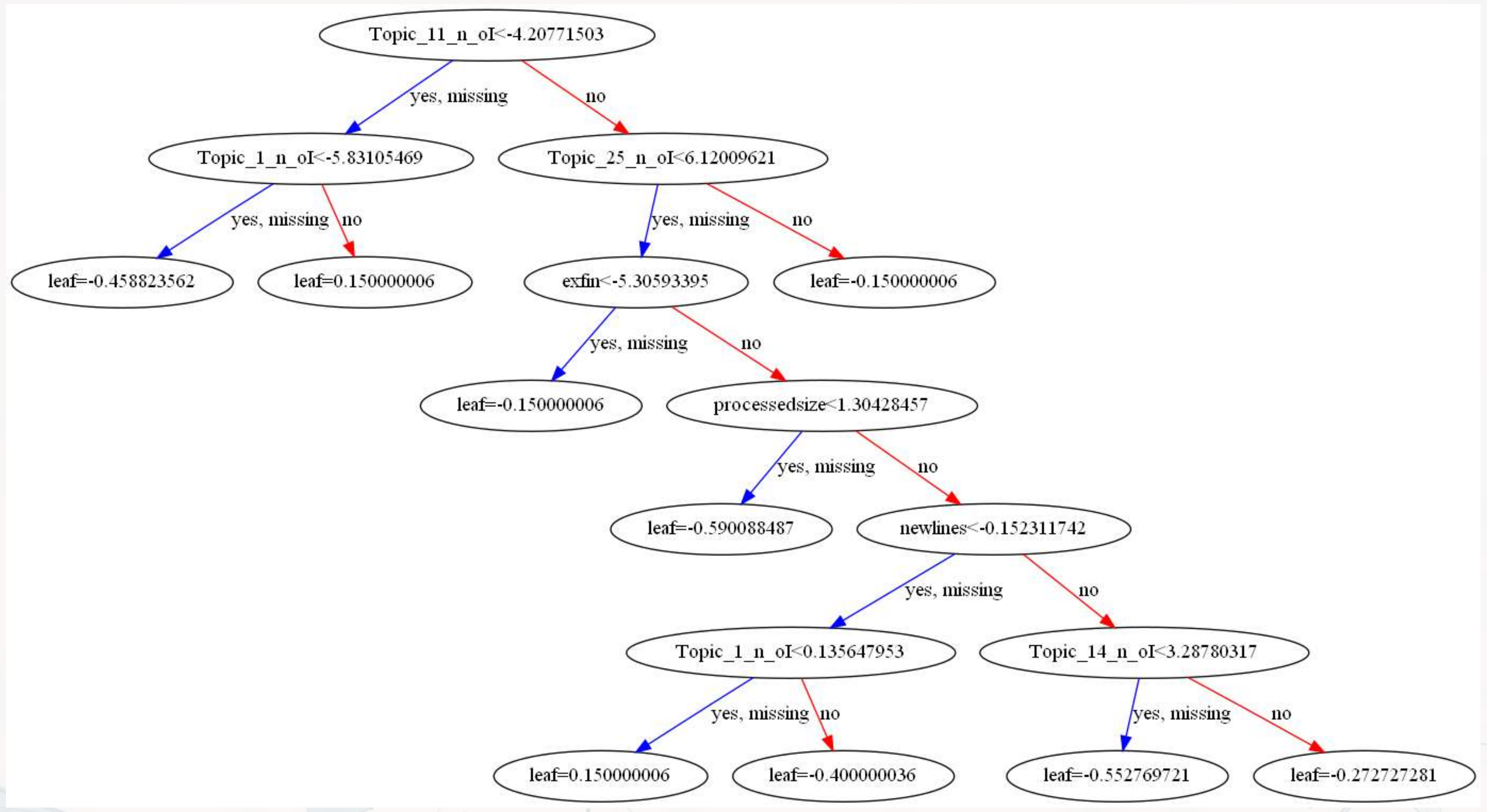
Analyzing the model: Importance plot

- 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

```python
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

What about optimizing all the parameters?

> This can be done – details are in the python code file

Using R to run XGBoost

- The same package, xgboost works for this in R
  - The level of support across R and python is more or less the same

XGBoost in python

- Can solve numeric problems well
- Can do GPU computations for some models
- Can run larger-than-memory computations
  - Good for big data sets!

XGBoost in R

- Can solve numeric problems well
- Can also handle categorical inputs

- Use tidymodels just like we did for SVM, but specify `tune()` for each parameter you want to tune

Running CV XGBoost in R

```r
# model setup
params <- list(max_depth=10,
               eta=0.2,
               gamma=10,
               min_child_weight = 5,
               objective =
                 "binary:logistic")

# run the model
xgbCV <- xgb.cv(params=params,
                data=train_x,
                label=train_y,
                nrounds=100,
                eval_metric="auc",
                nfold=10,
                stratified=TRUE)
```

Conclusion

Wrap-up

> SVM: Support Vector Machine

- Good for classification
- Can be good for regression in some contexts
- Key: Optimizes separability under some tolerance for error

> Tree models

- Strong classification performance
- Can handle sparsity well
- A somewhat interpretable yet non-linear class of models

Packages used for these slides

Python

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

R

- caret
- downlit
- kableExtra
- knitr
- quarto
- reticulate
- revealjs
- tidymodels
- tidyverse
- xgboost
- yardstick

References

- Chen, Tianqi, and Carlos Guestrin. “Xgboost: A scalable tree boosting system.” In Proceedings of the 22nd ACM sigkdd international conference on knowledge discovery and data mining, pp. 785-794. 2016.
- McInnes, Leland, John Healy, and James Melville. “UMAP: Uniform manifold approximation and projection for dimension reduction.” arXiv preprint arXiv:1802.03426 (2018).
- Mullainathan, Sendhil, and Jann Spiess. “Machine learning: an applied econometric approach.” Journal of Economic Perspectives 31, no. 2 (2017): 87-106.
- Noh, Byungjoo, Changhong Youm, Eunkyoung Goh, Myeounggon Lee, Hwayoung Park, Hyojeong Jeon, and Oh Yoen Kim. “XGBoost based machine learning approach to predict the risk of fall in older adults using gait outcomes.” Scientific reports 11, no. 1 (2021): 12183.
- Purda, Lynnette, and David Skillicorn. “Accounting variables, deception, and a bag of words: Assessing the tools of fraud detection.” Contemporary Accounting Research 32, no. 3 (2015): 1193-1223.

Custom code

Replication of R’s coefplot function for use with sklearn’s linear and logistic LASSO

```python
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=(4,0.25*len(data)))
    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
```