Beware Default Random Forest Importances

Brought to you by explained.ai

Terence Parr, Kerem Turgutlu, Christopher Csiszar, and Jeremy Howard
March 26, 2018.

(Terence is a tech lead at Google and ex-Professor of computer/data science; both he and Jeremy teach in University of San Francisco's MS in Data Science program. You might know Terence as the creator of the ANTLR parser generator. For more material, see Jeremy's fast.ai courses. Kerem and Christopher are current MS Data Science students.)

Update June 8, 2020. Terence (with James D. Wilson and Jeff Hamrick) just released Nonparametric Feature Impact and Importance that doesn't require a user's fitted model to compute impact. It's based upon a technique that computes Partial Dependence through Stratification.

Update July 18, 2019. scikit-learn just merged an implementation of permutation importance.

Update October 20, 2018 to show better feature importance plot and a new feature dependence heatmap. Updated all plots and section Dealing with collinear features. See new section Breast cancer data set multi-collinearities.

Updated April 19, 2018 to include new rfpimp package features to handle collinear dataframe columns in Dealing with collinear features section.

Updated April 4, 2018 to include many more experiments in the Experimental results section.

TL;DR

The scikit-learn Random Forest feature importance and R's default Random Forest feature importance strategies are biased. To get reliable results in Python, use permutation importance, provided here and in our rfpimp package (via `pip`). For R, use `importance=T` in the Random Forest

constructor then `type=1` in R's `importance()` function. In addition, your feature importance measures will only be reliable if your model is trained with suitable hyper-parameters.

Contents

Introduction to Feature Importance

Training a model that accurately predicts outcomes is great, but most of the time you don't just need predictions, you want to be able to *interpret* your model. For example, if you build a model of house prices, knowing which features are most predictive of price tells us which features people are willing to pay for. Feature importance is the most useful interpretation tool, and data scientists regularly examine model parameters (such as the coefficients of linear models), to identify important features.

Feature importance is available for more than just linear models. Most random Forest (RF) implementations also provide measures of feature importance. In fact, the RF importance technique we'll introduce here (*permutation importance*) is applicable to any model, though few machine learning practitioners seem to realize this. Permutation importance is a common, reasonably efficient, and very reliable technique. It directly measures variable importance by observing the effect on model accuracy of randomly shuffling each predictor variable. This technique is broadly-applicable because it doesn't rely on internal model parameters, such as linear regression coefficients (which are really just poor proxies for feature importance).

We recommend using permutation importance for all models, including linear models, because we can largely avoid any issues with model parameter interpretation. Interpreting regression coefficients requires great care and expertise; landmines include not normalizing input data, properly interpreting coefficients when using Lasso or Ridge regularization, and avoiding highly-correlated variables (such as country and country_name). To learn more about the difficulties of interpreting regression coefficients, see Statistical Modeling: The Two Cultures (2001) by Leo Breiman (co-creator of Random Forests).

One of Breiman's issues involves the accuracy of models. The more accurate our model, the more we can trust the importance measures and other interpretations. Measuring linear model goodness-of-fit is typically a matter of residual analysis. (A residual is the difference between predicted and expected outcomes). The problem is that residual analysis does not always tell us when the model is biased. Breiman quotes William Cleveland, "*one of the fathers of residual analysis,*" as saying residual analysis is an unreliable goodness-of-fit measure beyond four or five variables.

If a feature importance technique well-known to Random Forest implementers gives direct and reliable results, why have we written an article entitled “Beware Default Random Forest Importances?”

Trouble in paradise

Have you ever noticed that the feature importances provided by scikit-learn's Random Forests™ seem a bit off, perhaps not jiving with your domain knowledge? We've got some bad news—you can't always trust them. It's time to revisit any business or marketing decisions you've made based upon the default feature importances (e.g., which customer attributes are most predictive of sales). This is not a bug in the implementation, but rather an inappropriate algorithm choice for many data sets, as we discuss below. First, let's take a look at how we stumbled across this problem.

To prepare educational material on regression and classification with Random Forests (RFs), we pulled data from Kaggle's Two Sigma Connect: Rental Listing Inquiries competition and selected a few columns. Here are the first three rows of data in our data frame, `df`, loaded from data file rent.csv (`interest_level` is the number of inquiries on the website):

bathrooms	bedrooms	price	longitude	latitude	interest_level
1.5	3	3000	-73.942	40.714	2
1.0	2	5465	-73.966	40.794	1
1.0	1	2850	-74.001	40.738	3

We trained a regressor to predict New York City apartment rent prices using four apartment features in the usual scikit way:

```
features = ['bathrooms', 'bedrooms', 'longitude', 'latitude', 'price']
dfr = df[features]
X_train, y_train = dfr.drop('price',axis=1), dfr['price']
X_train['random'] = np.random.random(size=len(X_train))
rf = RandomForestRegressor(
        n_estimators=100,
        min_samples_leaf=1,
```

```
        n_jobs=-1,
        oob_score=True)
rf.fit(X_train, y_train)
```

In order to explain feature selection, we added a column of random numbers. (Any feature less important than a random column is junk and should be tossed out.)

After training, we plotted the `rf.feature_importances_` as shown in **Figure 1(a)**. Wow! New Yorkers really care about bathrooms. The number of bathrooms is the strongest predictor of rent price. That's weird but interesting.

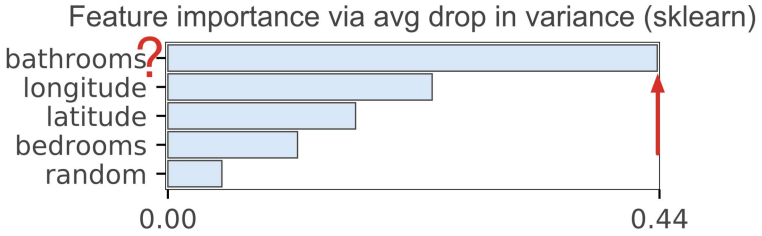


Figure 1(a). `scikit-learn` default importances for Random Forest **regressor** predicting apartment rental price from 4 features + a column of random numbers. Random column is last, as we would expect but the importance of the number of bathrooms for predicting price is highly suspicious.

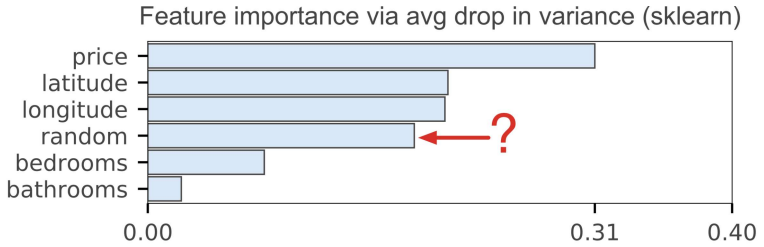


Figure 1(b). `scikit-learn` default importances for Random Forest **classifier** predicting apartment interest level (low, medium, high) using 5 features + a column of random numbers. Highly suspicious that random column is much more important than the number of bedrooms.

As expected, **Figure 1(a)** shows the random column as the least important.

Next, we built an RF classifier that predicts `interest_level` using the other five features and plotted the importances, again with a random column:

```
features = ['bathrooms', 'bedrooms', 'price', 'longitude', 'latitude', 'interest_level']
dfc = df[features]
X_train, y_train = dfc.drop('interest_level',axis=1), dfc['interest_level']
X_train['random'] = np.random.random(size=len(X_train))
rf = RandomForestClassifier(
        n_estimators=100,
        # better generality with 5
        min_samples_leaf=5,
        n_jobs=-1,
```

```
        oob_score=True)
rf.fit(X_train, y_train)
```

Figure 1(b) shows that the RF classifier thinks that the random column is more predictive of the interest level than the number of bedrooms and bathrooms. What the hell? Ok, something is definitely wrong.

Default feature importance mechanism

The most common mechanism to compute feature importances, and the one used in scikit-learn's RandomForestClassifier and RandomForestRegressor, is the *mean decrease in impurity* (or *gini importance*) mechanism (check out the Stack Overflow conversation). The mean decrease in impurity importance of a feature is computed by measuring how effective the feature is at reducing uncertainty (classifiers) or variance (regressors) when creating decision trees within RFs. The problem is that this mechanism, while fast, does not always give an accurate picture of importance. Breiman and Cutler, the inventors of RFs, indicate that this method of "*adding up the gini decreases for each individual variable over all trees in the forest gives a **fast** variable importance that is **often very consistent** with the permutation importance measure.*" (Emphasis ours and we'll get to permutation importance shortly.)

We've known for years that this common mechanism for computing feature importance is biased; i.e. it tends to inflate the importance of continuous or high-cardinality categorical variables For example, in 2007 Strobl *et al* pointed out in Bias in random forest variable importance measures: Illustrations, sources and a solution that "*the variable importance measures of Breiman's original Random Forest method ... are not reliable in situations where potential predictor variables vary in their scale of measurement or their number of categories.*" That's unfortunate because not having to normalize or otherwise futz with predictor variables for Random Forests is very convenient.

Permutation importance

Breiman and Cutler also described *permutation importance*, which measures the importance of a feature as follows. Record a baseline accuracy (classifier) or R^2 score (regressor) by passing a validation set or the out-of-bag (OOB) samples through the Random Forest. Permute the column values of a single predictor feature and then pass all test samples back through the Random Forest and recompute the accuracy or R^2. The importance of that feature is the difference between the baseline and the drop in overall accuracy or R^2 caused by permuting the column. The permutation mechanism is much more computationally expensive than the mean decrease in impurity mechanism, but the results are more reliable. The permutation importance strategy does not require retraining the model after permuting each column; we just have to re-run the perturbed test samples through the already-trained model.

Any machine learning model can use the strategy of permuting columns to compute feature importances. This fact is under-appreciated in academia and industry. Most software packages calculate feature importance using model parameters if possible (e.g., the coefficients in linear regression as discussed above). A single importance function could cover all models. The advantage of Random Forests, of course, is that they provide OOB samples by construction so users don't have to extract their own validation set and pass it to the feature importance function.

As well as being broadly applicable, the implementation of permutation importance is simple—here is a complete working function:

```
def permutation_importances(rf, X_train, y_train, metric):
    baseline = metric(rf, X_train, y_train)
    imp = []
    for col in X_train.columns:
        save = X_train[col].copy()
        X_train[col] = np.random.permutation(X_train[col])
        m = metric(rf, X_train, y_train)
        X_train[col] = save
        imp.append(baseline - m)
    return np.array(imp)
```

Notice that the function does not normalize the importance values, such as dividing by the standard deviation. According to Conditional variable importance for random forests, "*the raw [permutation] importance... has better statistical properties.*" Those importance values will not sum up to one and it's important to remember that we don't care what the values are *per se*. What we care about is the relative predictive strengths of the features. (When using the `importances()` function in R, make sure to use `scale=F` to to prevent this normalization.)

The key to this "baseline minus drop in performance metric" computation is to use a validation set or the OOB samples, not the training set (for the same reason we measure model generality with a validation set or OOB samples). Our `permutation_importances()` function expects the `metric` argument (a function) to use out-of-bag samples when computing accuracy or R^2 because there is no validation set argument. (We figured out how to grab the OOB samples from the scikit RF source code.) You can check out our functions that compute the OOB classifier accuracy and OOB regression R^2 score (without altering the RF model state). Here are two code snippets that call the permutation importance function for regressors and classifiers:

```
rf = RandomForestRegressor(...)
rf.fit(X_train, y_train) # rf must be pre-trained
imp = permutation_importances(rf, X_train, y_train,
                              oob_regression_r2_score)
```

```
rf = RandomForestClassifier(...)
imp = permutation_importances(rf, X_train, y_train,
                              oob_classifier_accuracy)
```

To test permutation importances, we plotted the regressor and classifier importances, as shown in **Figure 2(a)** and **Figure 2(b)**, using the same models from above. Both models included a random column, which correctly show up as the least important feature. The regressor in **Figure 1(a)** also had the random column last, but it showed the number of bathrooms as the strongest predictor of apartment rent price. The permutation importance in **Figure 2(a)** places bathrooms more reasonably as the least important feature, other than the random column.

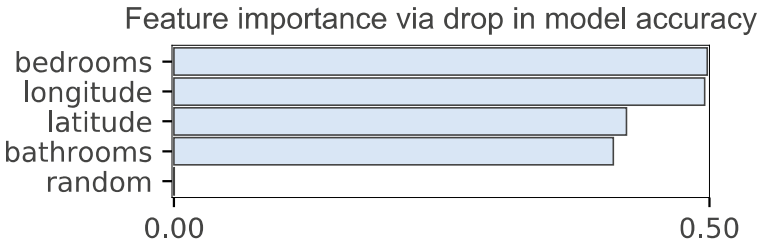


Figure 2(a). Importances derived by permuting each column and computing change in out-of-bag R^2 using `scikit-learn` **regressor**. Predicting apartment rental price from 4 features + a column of random numbers.

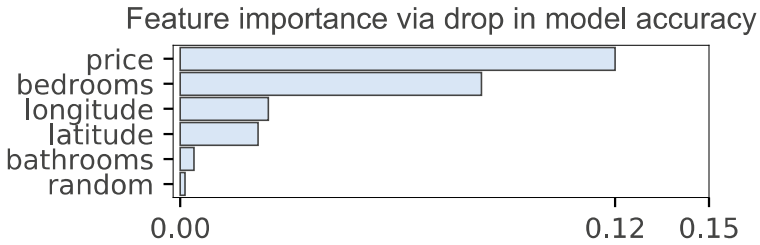


Figure 2(b). Importances derived by permuting each column and computing change in out-of-bag accuracy using `scikit-learn` Random Forest **classifier**.

The classifier default importances in **Figure 1(b)** are plausible, because price and location matter in real estate market. Unfortunately, the importance of the random column is in the middle of the pack, which makes no sense. **Figure 2(b)** places the permutation importance of the random column last, as it should be. One could also argue that the number of bedrooms is a key indicator of interest in an apartment, but the default mean-decrease-in-impurity gives the bedrooms feature little weight. The permutation importance in **Figure 2(b)**, however, gives a better picture of relative importance.

Permutation importance is pretty efficient and generally works well, but Strobl *et al* show that "*permutation importance over-estimates the importance of correlated predictor variables.*" in Conditional variable importance for random forests. It's unclear just how big the bias towards correlated predictor variables is, but there's a way to check.

Drop-column importance

Permutation importance does not require the retraining of the underlying model in order to measure the effect of shuffling variables on overall model accuracy. Because training the model can be extremely expensive and even take days, this is a big performance win. The risk is a potential bias towards correlated predictive variables.

If we ignore the computation cost of retraining the model, we can get the most accurate feature importance using a brute force *drop-column importance* mechanism. The idea is to get a baseline

performance score as with permutation importance but then drop a column entirely, retrain the model, and recompute the performance score. The importance value of a feature is the difference between the baseline and the score from the model missing that feature. This strategy answers the question of how important a feature is to overall model performance even more directly than the permutation importance strategy.

If we had infinite computing power, the drop-column mechanism would be the default for all RF implementations because it gives us a "ground truth" for feature importance. We can mitigate the cost by using a subset of the training data, but drop-column importance is still extremely expensive to compute because of repeated model training. Nonetheless, it's an excellent technique to know about and is a way to test our permutation importance implementation. The importance values could be different between the two strategies, but the order of feature importances should be roughly the same.

The implementation of drop-column is a straightforward loop like the permutation implementation and works with any model. For Random Forests, we don't need a validation set, nor do we need to directly capture OOB samples for performance measurement. In this case, we are retraining the model and so we can directly use the OOB score computed by the model itself. Here is the complete implementation:

```
def dropcol_importances(rf, X_train, y_train):
    rf_ = clone(rf)
    rf_.random_state = 999
    rf_.fit(X_train, y_train)
    baseline = rf_.oob_score_
    imp = []
    for col in X_train.columns:
        X = X_train.drop(col, axis=1)
        rf_ = clone(rf)
        rf_.random_state = 999
        rf_.fit(X, y_train)
        o = rf_.oob_score_
        imp.append(baseline - o)
    imp = np.array(imp)
    I = pd.DataFrame(
```

```
            data={'Feature':X_train.columns,
                  'Importance':imp})
    I = I.set_index('Feature')
    I = I.sort_values('Importance', ascending=True)
    return I
```

Notice that we force the `random_state` of each model to be the same. For the purposes of creating a general model, it's generally not a good idea to set the random state, except for debugging to get reproducible results. In this case, however, we are specifically looking at changes to the performance of a model after removing a feature. By controlling the random state, we are controlling a source of variability. Any change in performance should be due specifically to the drop of a feature.

Figure 3(a) and **Figure 3(b)** plot the feature importances for the same RF regressor and classifier from above, again with a column of random numbers. These results fit nicely with our understanding of real estate markets. Also notice that the random feature has negative importance in both cases, meaning that removing it improves model performance.

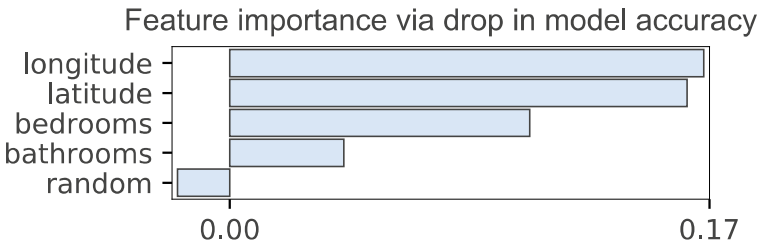


Figure 3(a). Importances derived by dropping each column, retraining `scikit-learn` Random Forest **regressor**, and computing change in out-of-bag R^2. Predicting apartment rental price from 4 features + a column of random numbers. The importance of the random column is at the bottom as it should be.

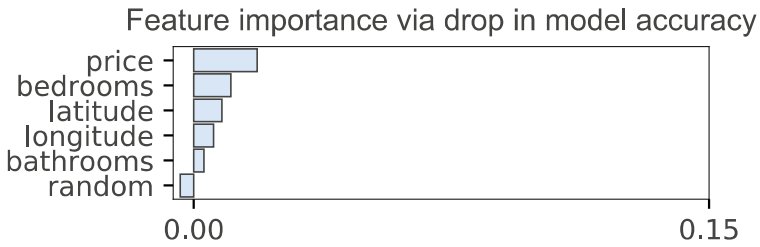


Figure 3(b). Importances derived by dropping each column, retraining `scikit-learn` Random Forest **classifier**, and computing change in out-of-bag accuracy. Predicting apartment interest level (low, medium, high) using 5 features + a column of random numbers. The importance of the random column is at the bottom as it should be.

That settles it for Python, so let's take a look at R, another popular language used for machine learning.

Comparing R to scikit-learn importances

Unfortunately, R's default importance strategy is mean-decrease-in-impurity, just like scikit, and so results are again unreliable. The reason for this default is that permutation importance is slower to compute than mean-decrease-in-impurity. For example, here's a code snippet (mirroring our Python code) to create a Random Forest and get the feature importances that traps the unwary:

```
# Warning! default is mean-decrease-in-impurity!
rf <- randomForest(price~., data = df[, 1:5], mtry=4, ntree = 40)
imp <- importance(rf)
```

To get reliable results, we have to turn on `importance=T` in the Random Forest constructor function, which then computes both mean-decrease-in-impurity and permutation importances. After that, we have to use `type=1` (not `type=2`) in the `importances()` function call:

```
rf <- randomForest(price~., data = df, mtry = 4, ntree = 40, importance=T)
imp <- importance(rf, type=1, scale = F) # permutation importances
```

Make sure that you don't use the `MeanDecreaseGini` column in the importance data frame; you want column `MeanDecreaseAccuracy`.

It's worth comparing R and scikit in detail. It not only gives us another opportunity to verify the results of our homebrewed permutation implementation, but we can also demonstrate that R's default type=2 importances have the same issues as scikit's only importance implementation.

R mean-decrease-in-impurity importance

R's mean-decrease-in-impurity importance (type=2) gives the same implausible results as we saw with scikit. To demonstrate this, we trained an RF regressor and classifier in R using the same data set and generated the importance graphs in **Figure 4**, which mirror the scikit graphs in **Figure 1**.

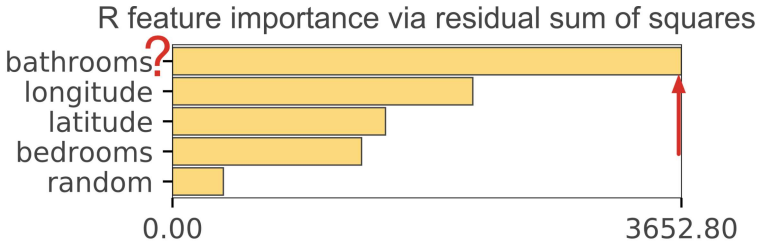


Figure 4(a). R's type=2 importances for Random Forest **regressor** predicting apartment rental price from 4 features + a column of random numbers. Random column is last, as we would expect but the importance of the number of bathrooms for predicting price is highly suspicious.

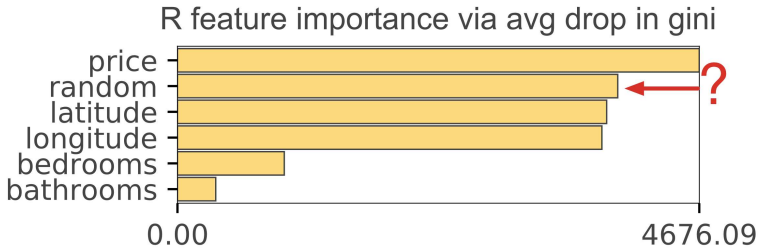


Figure 4(b). R's type=2 importances for Random Forest **classifier** predicting apartment interest level (low, medium, high) using 5 features + a column of random numbers. Highly suspicious that random column is much more important than the number of bedrooms.

R permutation importance

As a means of checking our permutation implementation in Python, we plotted and compared our feature importances side-by-side with those of R, as shown in **Figure 5** for regression and **Figure 6** for classification. The importance values themselves are different, but the feature order and relative levels are very similar, which is what we care about.

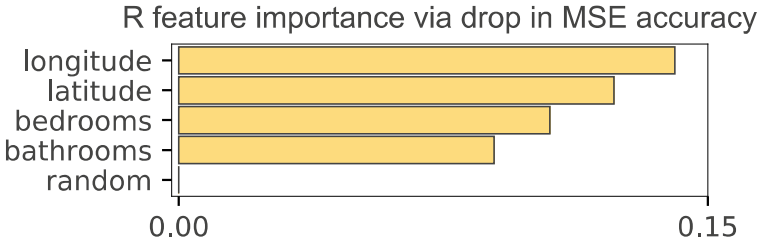


Figure 5(a). R's type=1 permutation importance for RF **regressor**.

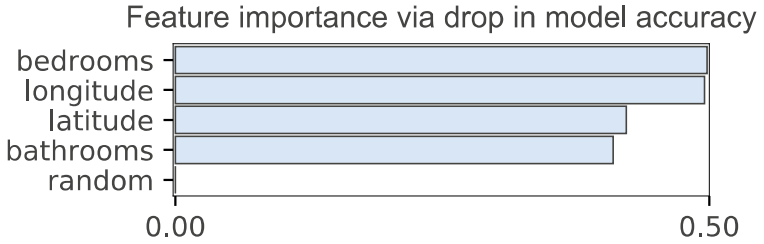


Figure 5(b). Python permutation importance for RF **regressor**

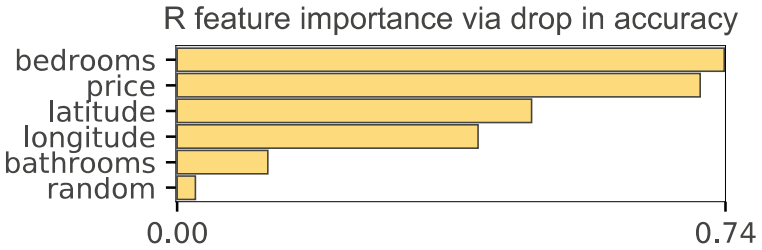


Figure 6(a). R's type=1 permutation importance for RF **classifier**.

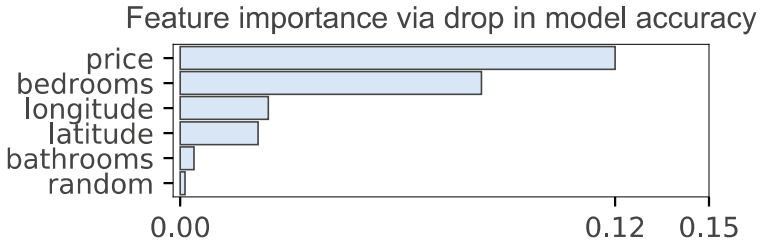


Figure 6(b). Python permutation importance for RF **classifier**.

R drop-column importance

For completeness, we implemented drop-column importance in R and compared it to our Python implementation, as shown in **Figure 8** for regression and **Figure 9** for classification.

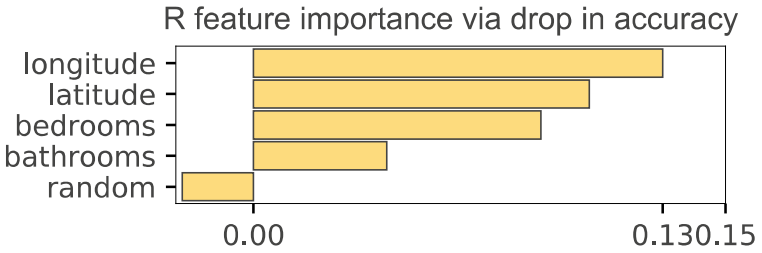


Figure 8(a). Importances derived by dropping each column, retraining an RF **regressor** in R, and computing the change in out-of-bag R^2.

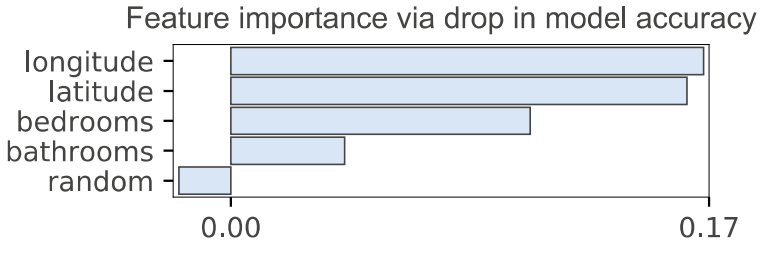


Figure 8(b). Importances derived by dropping each column, retraining a `scikit` RF **regressor**, and computing the change in out-of-bag R^2.

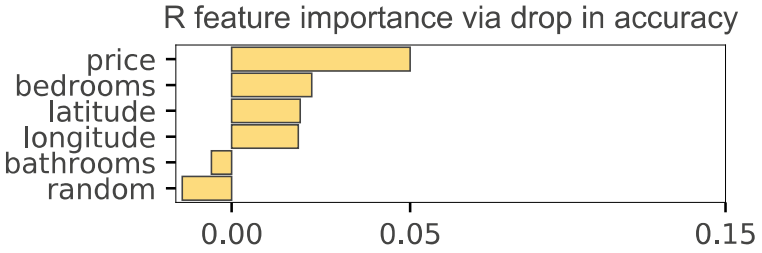


Figure 9(a). Importances derived by dropping each column, retraining an RF **classifier** in R, and computing the change in out-of-bag accuracy.

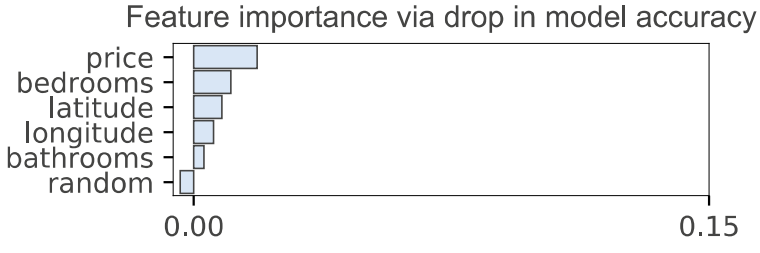


Figure 9(b). Importances derived by dropping each column, retraining a `scikit` RF **classifier**, and computing the change in out-of-bag accuracy.

Experimental results

Model-neutral permutation importance

The permutation importance code shown above uses out-of-bag (OOB) samples as validation samples, which limits its use to RFs. If we rely on the standard scikit `score()` function on models, it's a simple matter to alter the permutation importance to work on any model. Also, instead of passing in the training data, from which OOB samples are drawn, we have to pass in a validation set. (Don't

pass in your test set, which should only used as a final step to measure final model generality; the validation set is used to tune and probe a model.) Here's the core of the model-neutral version:

```
baseline = model.score(X_valid, y_valid)
imp = []
for col in X_valid.columns:
    save = X_valid[col].copy()
    X_valid[col] = np.random.permutation(X_valid[col])
    m = model.score(X_valid, y_valid)
    X_valid[col] = save
    imp.append(baseline - m)
```

See our function importances() in the rfpimp package.

Performance considerations

The use of OOB samples for permutation importance computation also has strongly negative performance implications. Using OOB samples means iterating through the trees with a Python loop rather than using the highly vectorized code inside scikit/numpy for making predictions. For even data sets of modest size, the permutation function described in the main body of this article based upon OOB samples is extremely slow.

On a (confidential) data set we have laying around with 452,122 training records and 36 features, OOB-based permutation importance takes about 7 minutes on a 4-core iMac running at 4Ghz with ample RAM. The invocation from a notebook in Jupyter Lab looks like:

```
from rfpimp import *
rf = RandomForestClassifier(n_estimators=100, n_jobs=-1)
%time I = oob_importances(rf, X_train, y_train)
```

Using a validation set with 36,039 records instead of OOB samples, takes about 8 seconds (`n_samples=-1` implies use all validation samples):

```
%time I = importances(rf, X_valid, y_valid, n_samples=-1)
```

If we further let the importances function use the default of 3,500 samples taken randomly from the validation set, the time drops to about 4 seconds. These test numbers are completely unscientific but give you a ballpark of speed improvement. 7 minutes down 4 seconds is pretty dramatic. (See the next section on validation set size.)

Using the much smaller rent.csv file, we see smaller durations overall but again using a validation set over OOB samples gives a nice boost in speed. With a validation set size 9660 x 4 columns (20% of the data), we see about 1 second to compute importances on the full validation set and 1/2 second using 3,500 validation samples.

We added a permutation importance function that computes the drop in accuracy using cross validation. Even for the small data set, the time cost of 32 seconds is prohibitive because of the retraining involved. Here's the invocation:

```
%time I = cv_importances(rf, X_train, y_train, k=5)
```

Similarly, the drop column mechanism takes 20 seconds:

```
%time I = dropcol_importances(rf, X_train, y_train)
```

It's faster than the cross validation because it is only doing a single training per feature not *k* trainings per feature.

We also looked at using the nice Eli5 library to compute permutation importances. Eli5's permutation mechanism also supports various kinds of validation set and cross validation strategies; the mechanism is also model neutral, even to models outside of scikit. On the confidential data set with 36,039 validation records, eli5 takes 39 seconds. On the smaller data set with 9660 validation records, eli5 takes 2 seconds. Here's the invocation we used:

```
from eli5.sklearn import PermutationImportance
%time perm = PermutationImportance(rf).fit(X_test, y_test)
```

The effect of validation set size on importance

We hypothesized that it's possible to extract meaningful feature importance without using a very large validation set, which would give us the opportunity to improve speed by using a subsample of the validation set. We ran simulations on two very different data sets, one of which is the rent data used in this article and the other is a 5x bigger confidential data set. **Figure 10** summarizes the results for the two data sets. From this we can conclude that 3500 is a decent default number of samples to use when computing importance using a validation set.

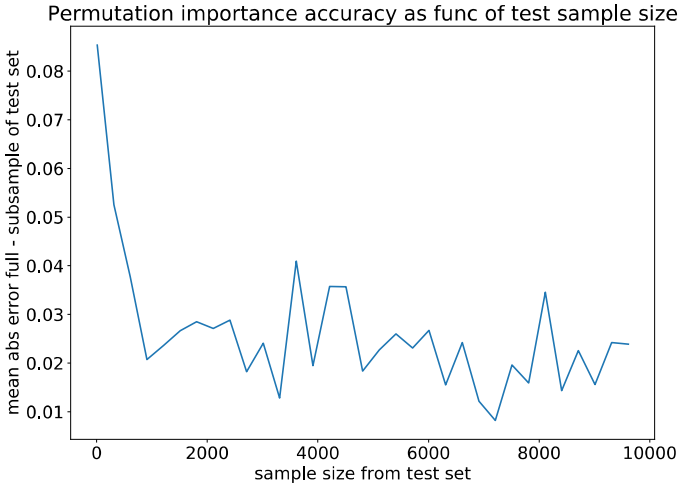


Figure 10(a). `rent.csv` data set. Mean absolute difference of feature importance values for a validation set subsample compared to feature important values computed using entire validation set. Regressor on 38,640 training records and 4 features; 9,660 validation records.

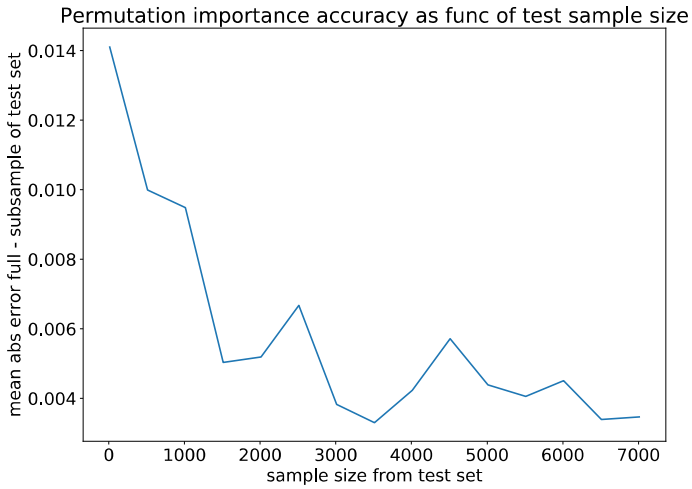


Figure 10(b). Confidential data set. Mean absolute difference of feature importance values for a validation set subsample compared to feature important values computed using entire validation set. Binary classifier on 452,122 training records and 36 features; 36,039 validation records.

Naturally, this is only two data sets and so the importances function takes a `n_samples` argument so you can experiment (-1 implies entire validation set.) Our rfpimp package is really meant as an educational exercise but you're welcome to use the library for actual work if you like. :)

The effect of collinear features on importance

[See Dealing with collinear features section below.]

Feature importance is a key part of model interpretation and understanding the business problem that originally drove you to create a model in the first place. We have to keep in mind, though, that the feature importance mechanisms we describe in this article consider each feature individually. If all features are totally independent and not correlated in any way, than computing feature importance individually is no problem. If, however, two or more features are *collinear* (correlated in some way but not necessarily with a strictly linear relationship) computing feature importance individually can give unexpected results.

The effect of collinear features is most stark when looking at drop column importance. **Figure 11(a)** shows the drop column importance on a decent regressor model (R^2 is 0.85) for the rent data. **Figure 11(b)** shows the exact same model but with the longitude column duplicated. At first, it's shocking to see the most important feature disappear from the importance graph, but remember that we measure importance as a drop in accuracy. If we have two longitude columns and drop one, there should not be a change in accuracy (at least for a RF model that doesn't get confused by duplicate columns.) Without a change in accuracy from the baseline, the importance for a dropped feature is zero.

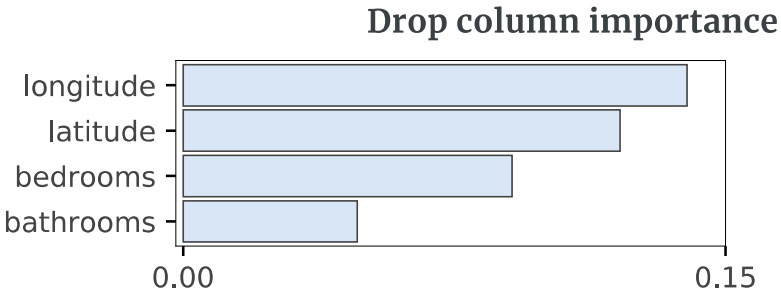


Figure 11(a). **Drop column importance** strategy using random forest regressor with R^2=0.85 (so a pretty decent predictor of rent price).

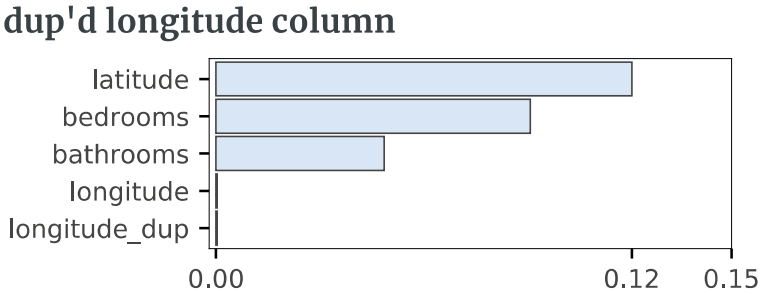


Figure 11(b). After duplicating `longitude` column and rerunning **drop column importance**. Both longitude columns disappear because removing one does not affect model accuracy at all.

It's also worth pointing out that feature importances should only be trusted with a strong model. If your model does not generalize accurately, feature importances are worthless. If your model is weak, you will notice that the feature importances fluctuate dramatically from run to run. That's why we mention the R^2 of our model.

The effect of collinear features on permutation importance is more nuanced and depends on the model; we'll only discuss RFs here. During decision tree construction, node splitting should choose equally important variables roughly 50-50. So, in a sense, conking the RF on the head with a coconut by permuting one of those equally important columns should be half supported by the other identical column during prediction. In fact, that's exactly what we see empirically in **Figure 12(b)** after duplicating the longitude column, retraining, and rerunning permutation importance.

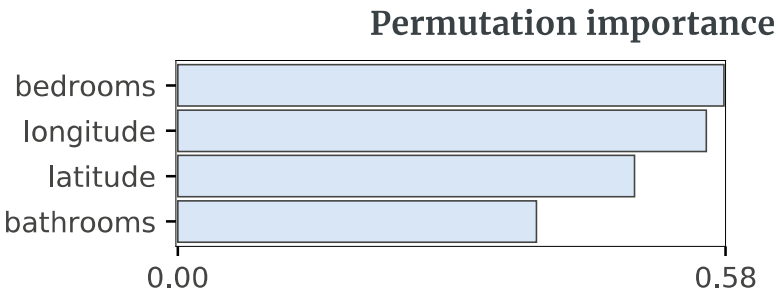


Figure 12(a). **Permutation importance** strategy using random forest regressor.

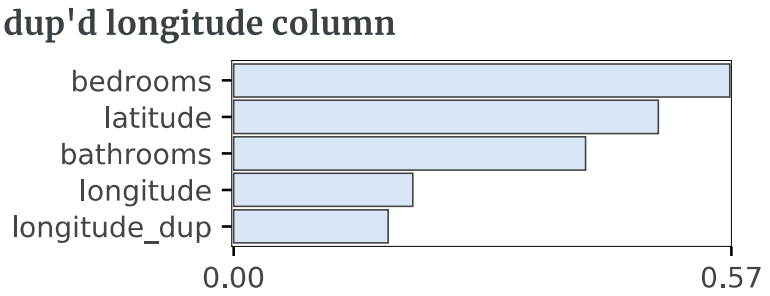


Figure 12(b). After duplicating `longitude` column and rerunning **permutation importance**. Both longitude columns share importance and so we see roughly 50-50 important.

When features are correlated but not duplicates, the importance should be shared roughly per their correlation (in the general sense of correlation, not the linear correlation coefficient). We did an experiment adding a bit of noise to the duplicated longitude column to see its effect on importance. The longitude range is 0.3938 so let's add uniform noise in range 0..c for some constant c that is somewhat smaller than that range:

```
noise = np.random.random(len(X_train_val)) * c
X_train['longitude_noisy'] = X_train.longitude + noise
```

With just a tiny bit of noise, c = .0005, **Figure 13(a)** shows the noisy longitude column pulling down the importance of the original longitude column. **Figure 13(b)** shows the importance graph with c = .001.

Permutation importance dup'd longitude + noise column

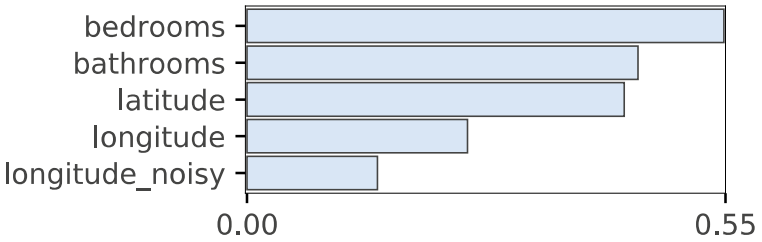


Figure 13(a). With uniform noise in range 0-0.0005, noisy longitude column pulls down the importance of the real column.

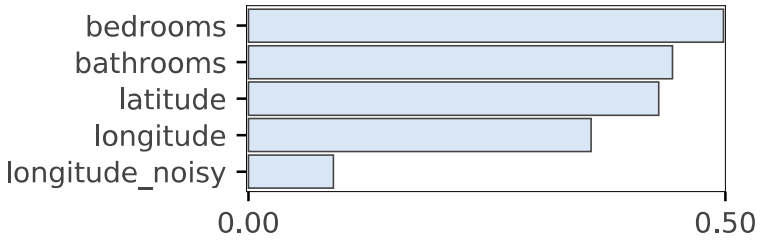


Figure 13(b). With uniform noise in range 0-0.001, noisy longitude column doesn't pull down the importance of the real column very much.

We performed the same experiment by adding noise to the bedrooms column, as shown in **Figure 14**. Notice that it chose the noisy column in **Figure 14(a)** as the most important, which happened by chance because they are so similar.

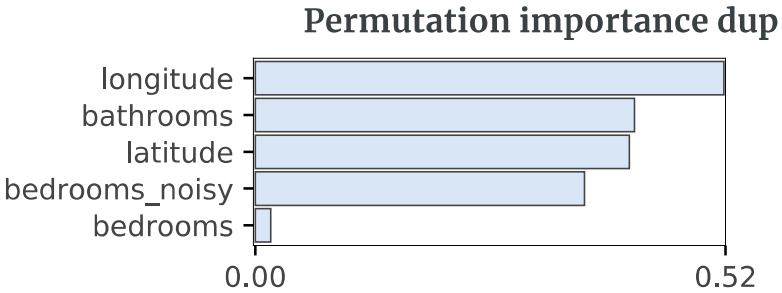


Figure 14(a). With uniform noise in range 0-1, noisy bedrooms column pulls down the importance of the real column.

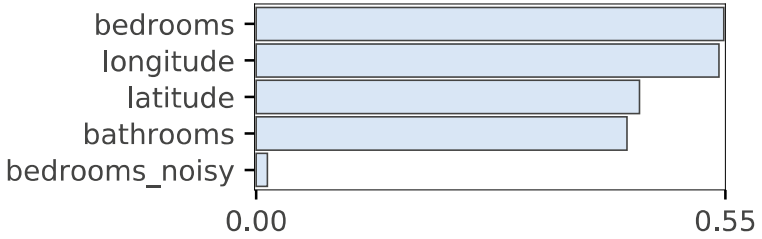


Figure 14(b). With uniform noise in range 0-3, noisy bedrooms column doesn't pull down the importance of the real column very much.

While we're at it, let's take a look at the effect of collinearity on the mean-decrease-in-impurity (gini importance). **Figure 15** illustrates the effect of adding a duplicate of the longitude column when using the default importance from scikit RFs. As with the permutation importance, the duplicated longitude column pulls down the importance of the original longitude column because it is sharing with the duplicated column.

Mean-decrease-in-impurity importance dup'd longitude column

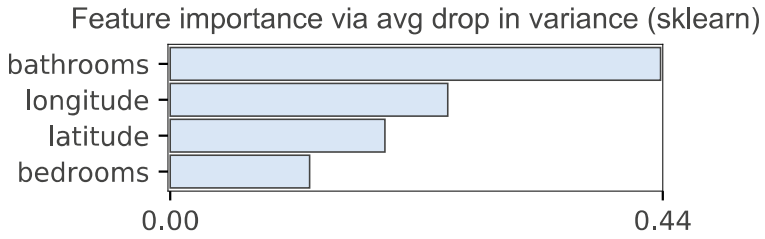


Figure 15(a). **Mean-decrease-in-impurity importance** strategy using random forest regressor.

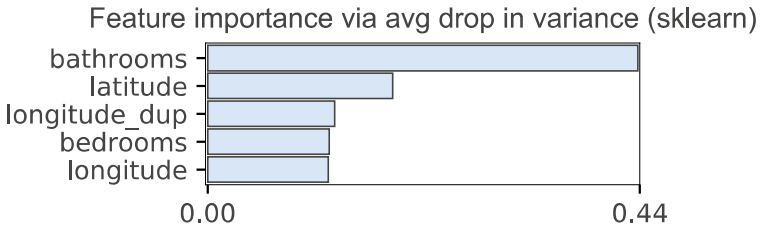


Figure 15(b). After duplicating longitude column and rerunning **mean-decrease-in-impurity importance**. Both longitude columns share importance roughly 50-50.

Adding a noisy duplicate of the longitude column behaves like permutation importance as well, stealing importance from the original longitude column in proportion to the amount of noise, as shown in **Figure 16**. Naturally, we still have the odd behavior that bathrooms is considered the most important feature.

Mean-decrease-in-impurity importance dup'd longitude + noise column

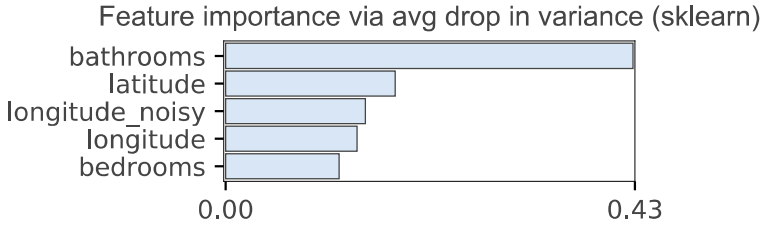


Figure 16(a). With uniform noise in range 0-0.0005, noisy longitude column pulls down the importance of the real column, just as with permutation importance.

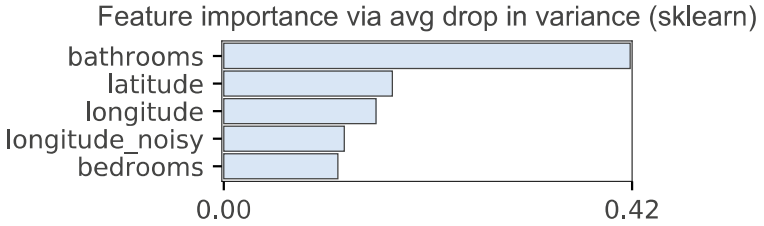


Figure 16(b). With uniform noise in range 0-0.001, noisy longitude column doesn't pull down the importance of the real column very much, just as with permutation importance.

From these experiments, it's safe to conclude that permutation importance (and mean-decrease-in-impurity importance) computed on random forest models spreads importance across collinear variables. The amount of sharing appears to be a function of how much noise there is in between the two. We do not give evidence that correlated, rather than duplicated and noisy variables, behave in the same way. On the other hand, one can imagine that longitude and latitude are correlated in some way and could be combined into a single feature. Presumably this would show twice the importance of the individual features.

You can find all of these collinearity experiments in collinear.ipynb

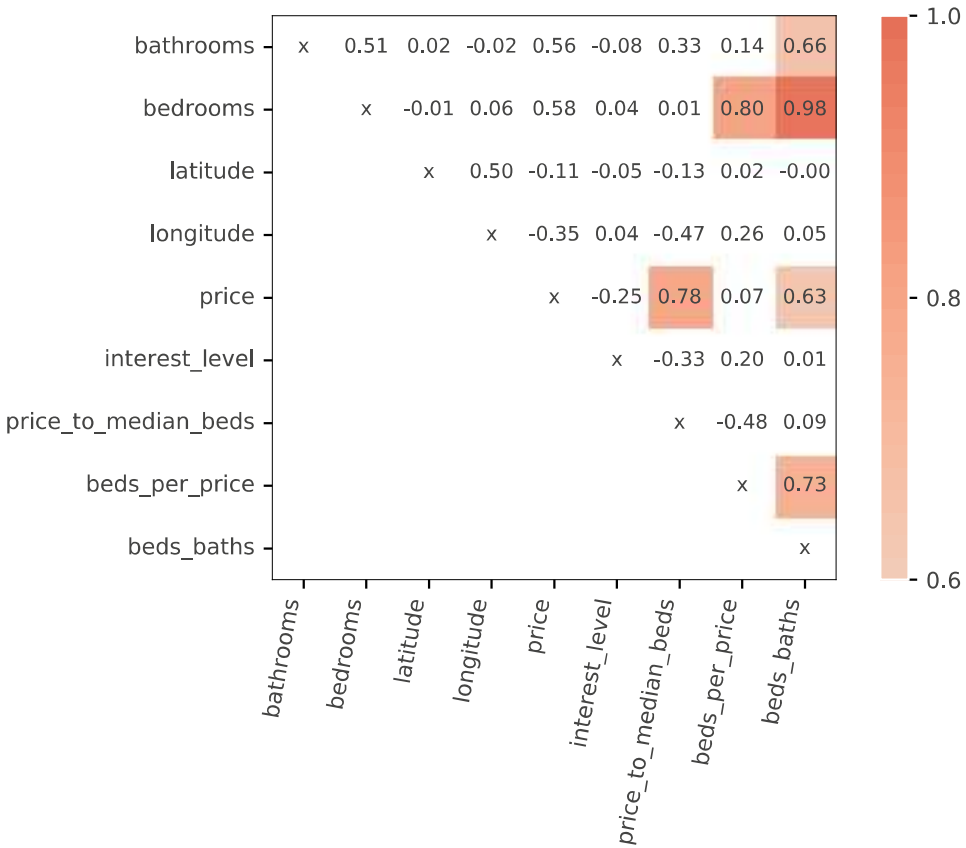
Dealing with collinear features

We updated the rfpimp package (1.1 and beyond) to help understand importance graphs in the presence of collinear variables. To get an understanding of collinearity between variables, we created `feature_corr_matrix(df)` that takes a data frame and returns the Spearman's rank-order correlation between all pairs of features as a matrix with feature names as index and column names. The result is a data frame in its own right. Here's a sample:

	bathrooms	bedrooms	latitude	longitude	price	interest_level	price_to_median_beds	beds_per_price	beds_baths
bathrooms	1.0000	0.5161	0.0156	-0.0233	0.5577	-0.0768	0.3304	0.1430	0.6611
bedrooms	0.5161	1.0000	-0.0075	0.0625	0.5765	0.0401	0.0086	0.8001	0.9786
latitude	0.0156	-0.0075	1.0000	0.5054	-0.1112	-0.0470	-0.1355	0.0267	-0.0049
longitude	-0.0233	0.0625	0.5054	1.0000	-0.3548	0.0315	-0.4728	0.2612	0.0474
price	0.5577	0.5765	-0.1112	-0.3548	1.0000	-0.2424	0.7837	0.0644	0.6249
interest_level	-0.0768	0.0401	-0.0470	0.0315	-0.2424	1.0000	-0.3302	0.1964	0.0145
price_to_median_beds	0.3304	0.0086	-0.1355	-0.4728	0.7837	-0.3302	1.0000	-0.4831	0.0894
beds_per_price	0.1430	0.8001	0.0267	0.2612	0.0644	0.1964	-0.4831	1.0000	0.7325
beds_baths	0.6611	0.9786	-0.0049	0.0474	0.6249	0.0145	0.0894	0.7325	1.0000

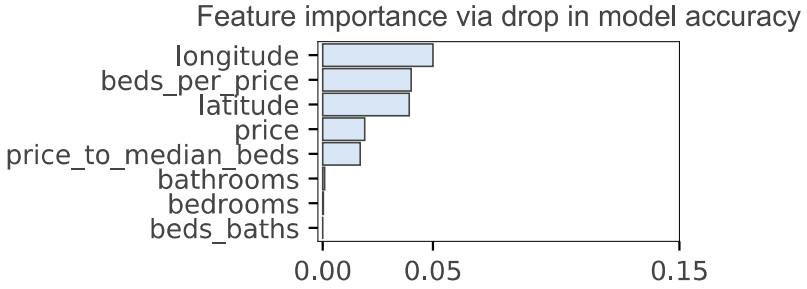
Spearman's correlation is the same thing as converting two variables to rank values and then running a standard Pearson's correlation on those ranked variables. Spearman's is nonparametric and does not assume a linear relationship between the variables; it looks for monotonic relationships. You can visualize this more easily using `plot_corr_heatmap()`:

```
from rfpimp import plot_corr_heatmap
viz = plot_corr_heatmap(df_train, figsize=(7,5))
viz.view()
```



Because it is a symmetric matrix, only the upper triangle is shown. The diagonal is all x's since auto-correlation is not useful.

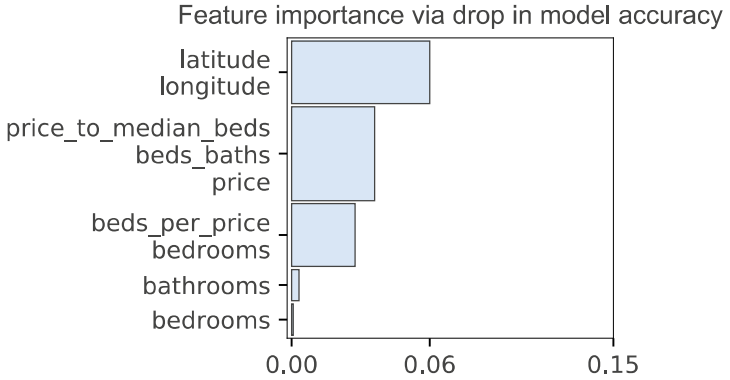
As we discussed, permutation feature importance is computed by permuting a specific column and measuring the decrease in accuracy of the overall classifier or regressor. Of course, features that are collinear really should be permuted together. We have updated `importances()` so you can pass in either a list of features, such as a subset, or a list of lists containing groups. Let's start with the default:



You can pass in a list with a subset of features interesting to you. All unmentioned features will be grouped together into a single meta-feature on the graph. You can also pass in a list that has sublists like: `[['latitude', 'longitude'], 'price', 'bedrooms']`. Each string or sublist will be permuted together as a feature or meta-feature; the drop in overall accuracy of the model is the relative importance. Consider the following list of features and groups of features and snippet.

```
features = ['bathrooms', 'bedrooms',
            ['latitude', 'longitude'],
            ['price_to_median_beds', 'beds_baths', 'price'],
            ['beds_per_price','bedrooms']]
I = importances(rf, X_test, y_test, features=features)
plot_importances(I)
```

Notice how, in the following result, latitude and longitude together are very important as a meta-feature. The meta-features "steal" importance from the individual bedrooms and bathrooms columns.



Bar thickness indicates the number of features in the group.

Any features not mentioned get lumped together into a single "other" meta-feature, so that all features are considered. Imagine a model with 10 features and we requested a feature importance graph with just two very unimportant features. He would look like one or the other were very important, which could be very confusing. So, the importance of the specified features is given only in comparison to all possible futures. **Figure 17** shows two different sets of features and how all others are lumped together as one meta-feature.

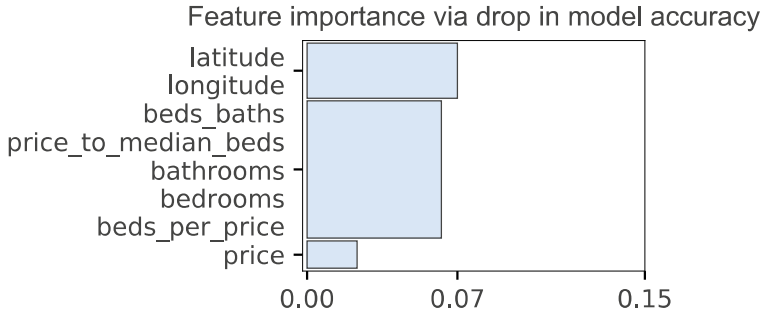


Figure 17(a). features=['price',['latitude','longitude']]

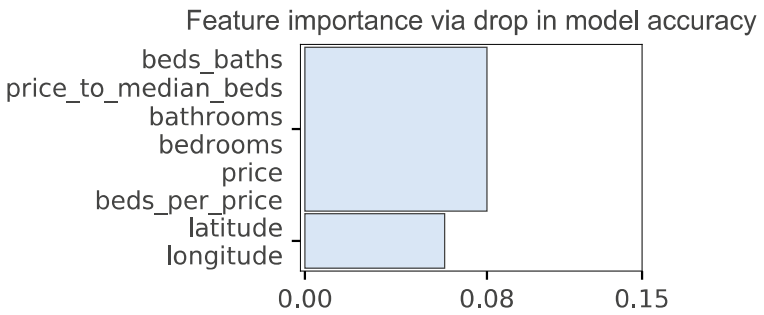
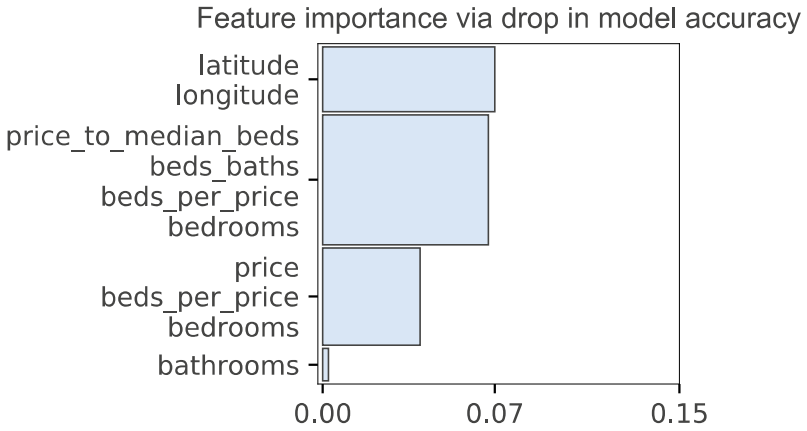


Figure 17(b). features=[['latitude','longitude']]

Features can also appear in multiple feature groups so that we can compare the relative importance of multiple meta-features that once. Remember that the permutation importance is just permuting all features associated with the meta-feature and comparing the drop in overall accuracy. There's no reason we can't do multiple overlapping sets of features in the same graph. For example, in the following, feature list, bedrooms appears in two meta-features as does `beds_per_price`.

```
features = [['latitude', 'longitude'],
            ['price_to_median_beds', 'beds_baths', 'beds_per_price', 'bedrooms'],
            ['price','beds_per_price','bedrooms']]
```



Breast cancer data set multi-collinearities

As another example, let's look at the techniques described in this article applied to the well-known breast cancer data set. There are 569 observations each with 30 numerical features and a single binary malignant/benign target variable. A random forest makes short work of this problem, getting about 95% accuracy using the out-of-bag estimate and a holdout testing set. The default feature importance computation from scikit-learn gives a beautiful graph and that biases us to consider it meaningful and accurate. On the other hand, if we look at the permutation importance and the drop column importance, no feature appears important.

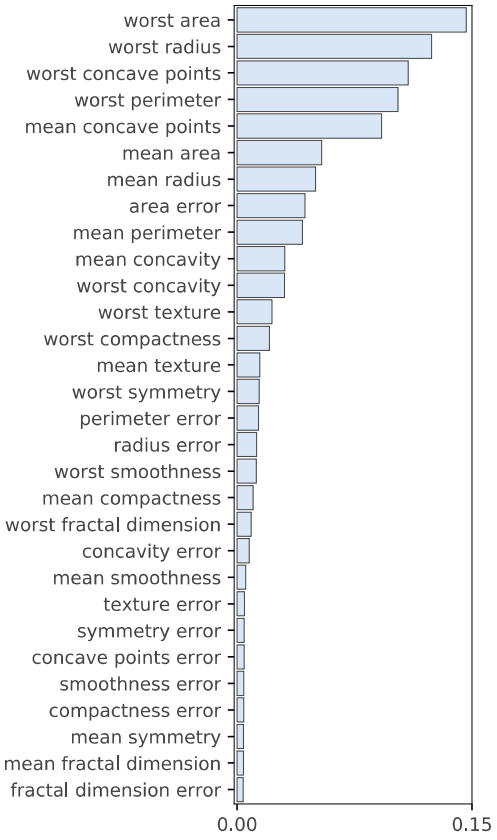


Figure 18(a). Default sci-kit learn average gini drop feature importance

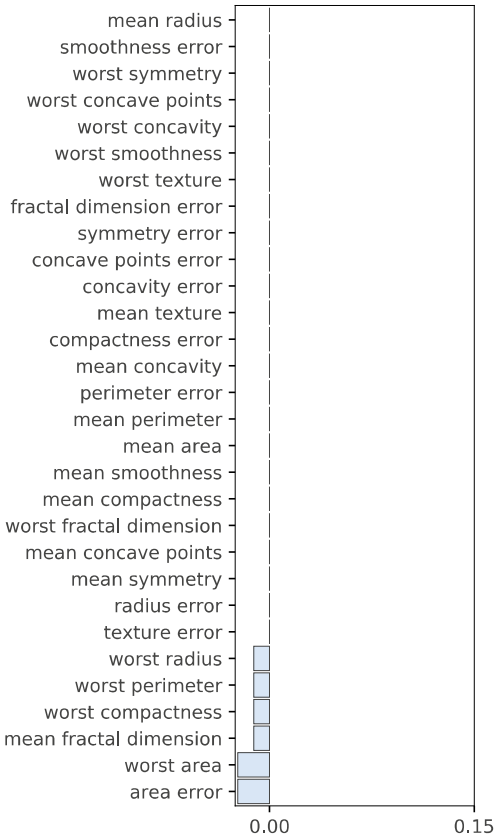


Figure 18(b). Permutation feature importance

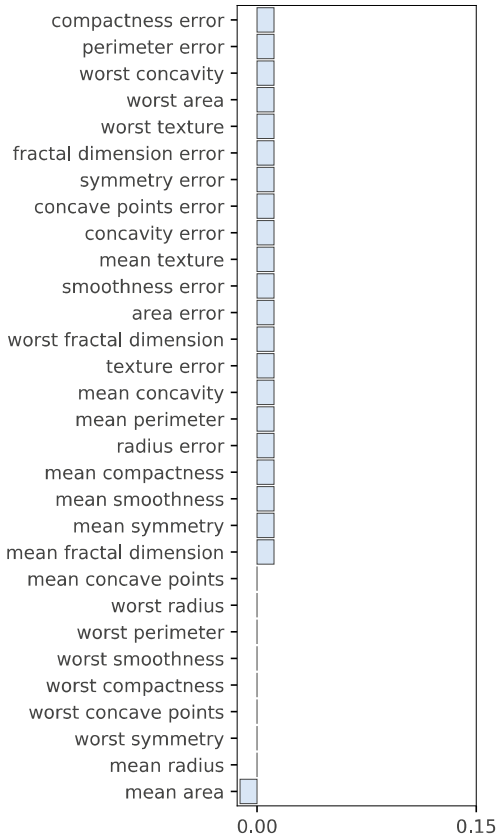


Figure 18(c). Drop column feature importance

At first, using default bar charts, it looked like the permutation importance was giving a signal. We get so focused on the relative importance we don't look at the absolute magnitude of the importance. The magnitude indicates the drop in classification accuracy or R^2 (regressors) and so it is meaningful. For that reason, our `plot_importances` function sets a minimum bound of 0.15 so that users notice when the feature importance is near zero or very low. When feature importances are very low, it either means the feature is not important or it is highly collinear with one or more other features.

A way to identify if a feature, x, is dependent on other features is to train a model using x as a dependent variable and all other features as independent variables (this is called Multicollinearity). Because random forests give us an easy out of bag error estimate, the feature dependence functions in `rfpimp` rely on random forest models. The R^2 prediction error from the model indicates how easy it is to predict feature x using the other features. The higher the score, the more dependent feature x is. The feature importance of non-x features predicting x give an indication of which features have predictive power for feature x. Compare the correlation and feature dependence heat maps (click to enlarge images):

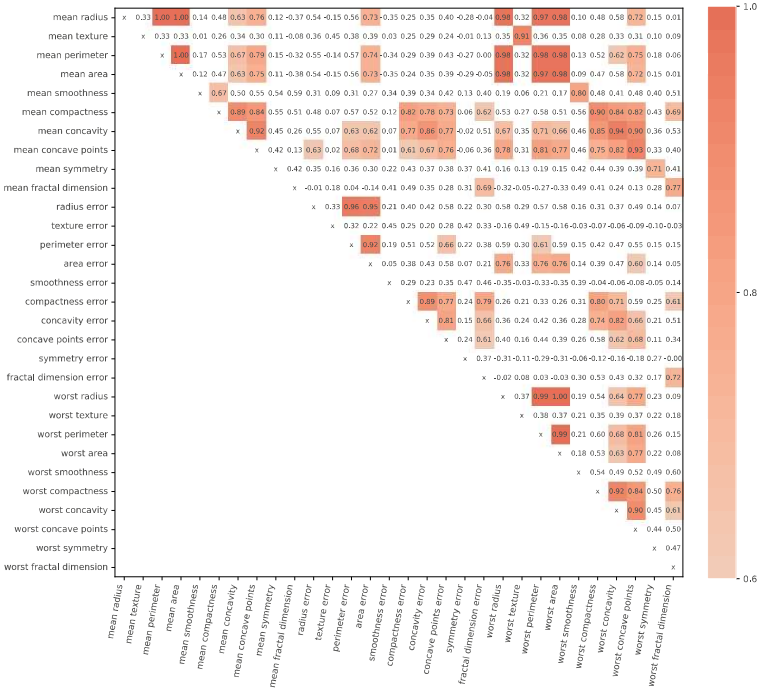


Figure 18(a). Spearman's correlation matrix between features i and j

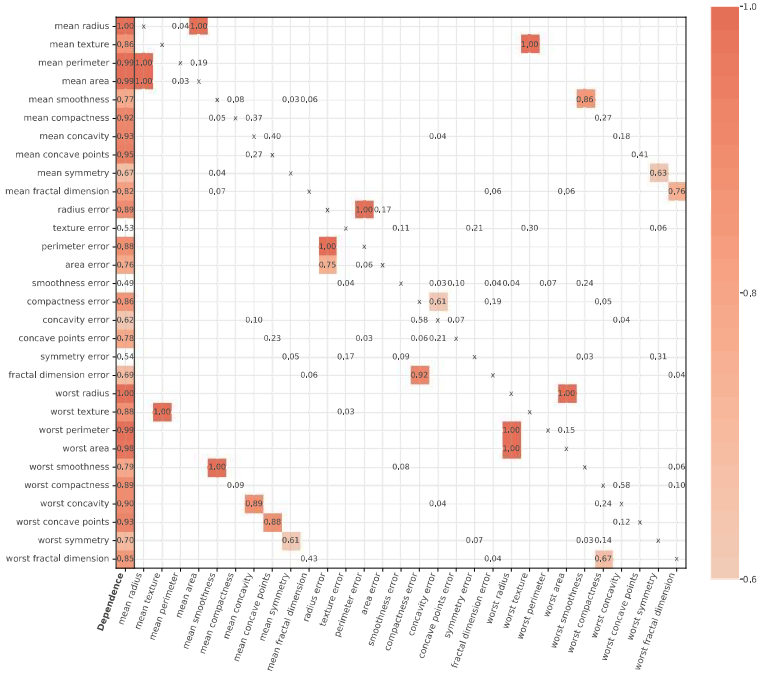


Figure 18(b). Feature dependence matrix computed by predicting feature i using all j!=i as predictor variables. Values of the matrix are the feature importances obtained from model predicting i.

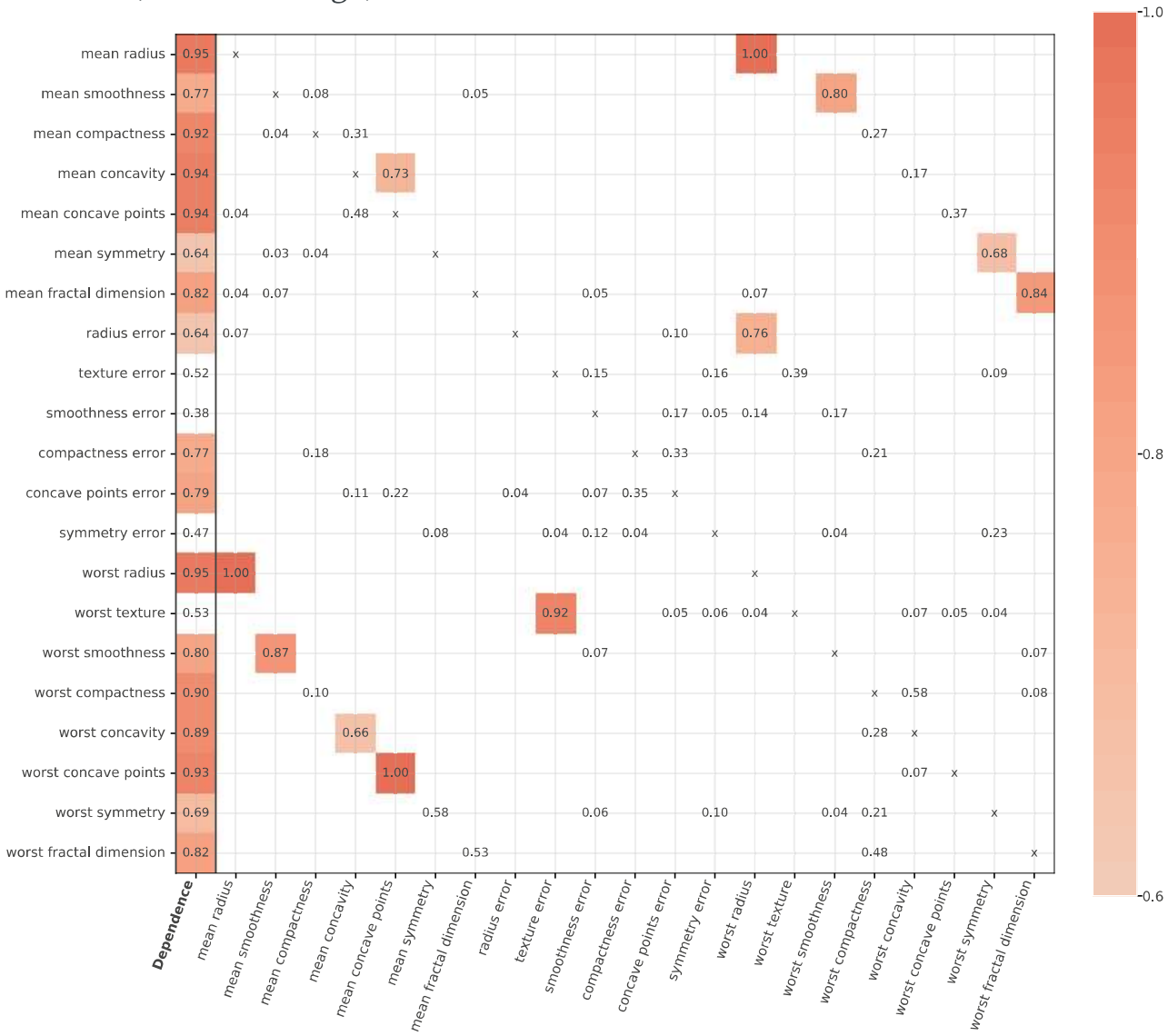
Here are the dependence measures for the various features (from the first column of the dependence matrix):

Feature	Dependence
mean radius	0.995
worst radius	0.995
mean perimeter	0.994
mean area	0.984
worst perimeter	0.983
worst area	0.978
radius error	0.953
mean concave points	0.944
mean concavity	0.936
worst concave points	0.927
mean compactness	0.916
worst concavity	0.901
perimeter error	0.898
worst compactness	0.894
worst texture	0.889
compactness error	0.866
mean texture	0.856
worst fractal dimension	0.84
area error	0.835
mean fractal dimension	0.829
concave points error	0.786
worst smoothness	0.764
mean smoothness	0.754
fractal dimension error	0.741
worst symmetry	0.687
mean symmetry	0.659
concavity error	0.623

texture error	0.514
smoothness error	0.483
symmetry error	0.434

Dependence numbers close to one indicate that the feature is completely predictable using the other features, which means it could be dropped without affecting accuracy. (Dropping features is a good idea because it makes it easier to explain models to consumers and also increases training and testing efficiency/speed.) For example, the mean radius is extremely important in predicting mean perimeter and mean area, so we can probably drop those two. It also looks like radius error is important to predicting perimeter error and area error, so we can drop those last two. Mean and worst texture also appear to be dependent, so we can drop one of those too. Similarly, let's drop concavity error and fractal dimension error because compactness error seems to predict them well. Worst radius also predicts worst perimeter and worst area well. Essentially, we're looking for columns with multiple entries close to 1.0 as those are the features that predict multiple other features. Dropping those 9 features has little effect on the OOB and test accuracy when modeled using a 100-tree random forest. Here's what the dependence matrix looks like without those

features (click to enlarge):



Keep in mind that low feature dependence does not imply unimportant. It just means that the feature is not collinear in some way with other features.

You can find all of these experiments trying to deal with collinearity in rfpimp-collinear.ipynb and pimp_plots.ipynb.

Summary

The takeaway from this article is that the most popular RF implementation in Python (scikit) and R's RF default importance strategy do not give reliable feature importances when "*... potential predictor variables vary in their scale of measurement or their number of categories.*" (Strobl *et al*). Rather than figuring out whether your data set conforms to one that gets accurate results, simply use permutation importance. You can either use our Python implementation (rfpimp via `pip`) or, if using R, make sure to use `importance=T` in the Random Forest constructor then `type=1` in R's `importance()` function.

Finally, we'd like to recommend the use of permutation, or even drop-column, importance strategies for all machine learning models rather than trying to interpret internal model parameters as proxies for feature importances.

Resources and sample code

- Breiman and Cutler are the inventors of RFs, so it's worth checking out their discussion of variable importance. They describe the mean-decrease-in-impurity importance and also the permutation importance.
- The eli5 library appears to be an excellent add-on to scikit-learn for visualizing and debugging. They have a permutation importance implemented as well.
- A good source of information on the bias associated with mean-decrease-in-impurity importance is Strobl *et al* from 2007: Bias in random forest variable importance measures: Illustrations, sources and a solution.
- To go beyond basic permutation importance, check out Strobl *et al*'s paper: Conditional variable importance for random forests. This article also talks about why we should use the raw mean decrease in accuracy score rather than normalizing it by dividing by the standard deviation.

- Ando Saabas has a nice blog entry called Selecting good features – Part III: random forests that includes an implementation of permutation importance, but it requires a validation set instead of using out-of-bag samples. The same author also has a blog describing stability selection and recursive feature implementation, related to this topic.

If your data set is not too big or you have a really beefy computer, you can always use the drop-column importance measure to get an accurate picture of how each variable affects the model performance.

Python

We produced a number of Jupyter notebooks to explore the issues described in this article, one for Python regressors and one for Python classifiers.

The overall github repo associated with this article has the notebooks and the source of a package you can install. You can explore the key (documented) functions directly in rfpimp.py or just install via pip:

```
$ pip install rfpimp
```

Here's an example using the rfpimp package to train a regressor, compute the permutation importances, and plot them in a horizontal bar chart:

```
from rfpimp import *
from rfpimp import *
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

df = pd.read_csv("/Users/parrt/github/random-forest-importances/notebooks/data/rent.csv")

# attentuate affect of outliers in price
df['price'] = np.log(df['price'])

df_train, df_test = train_test_split(df, test_size=0.20)
```

```
features = ['bathrooms','bedrooms','longitude','latitude',
            'price']
df_train = df_train[features]
df_test = df_test[features]

X_train, y_train = df_train.drop('price',axis=1), df_train['price']
X_test, y_test = df_test.drop('price',axis=1), df_test['price']
X_train['random'] = np.random.random(size=len(X_train))
X_test['random'] = np.random.random(size=len(X_test))

rf = RandomForestRegressor(n_estimators=100, n_jobs=-1)
rf.fit(X_train, y_train)

imp = importances(rf, X_test, y_test) # permutation
viz = plot_importances(imp)
viz.view()
```

R

We also created R Jupyter notebooks to explore these issues: R regressors and R classifiers.

Unlike scikit, R has a permutation importance implementation, but it's not the default behavior. Make sure that you don't use the `MeanDecreaseGini` column in the importance data frame. You want `MeanDecreaseAccuracy`, which only appears in the importance data frame if you turn on `importance=T` when constructing the Random Forest. The default when creating a Random Forest is to compute only the mean-decrease-in-impurity. Here's the proper invocation sequence:

```
rf <- randomForest(hyper-parameters..., importance=T)
imp <- importance(rf, type=1, scale = F) # permutation importances
```

Sample Kaggle apartment data

The data used by the notebooks and described in this article can be found in rent.csv, which is a subset of the data from Kaggle's Two Sigma Connect: Rental Listing Inquiries competition.

Epilogue: Explanations and Further Possibilities

It seems a shame that we have to choose between biased feature importances and a slow method. Can't we have both? And why is the decrease in gini method biased in the first place? Answering these questions requires more background in RF construction that we have time to go into right now, but here's a bit of a taste of an answer for those of you ready to do some further study.

In short, the answer is yes, we can have both. Extremely randomized trees, at least in theory, do not suffer from this problem. Better still, they're generally faster to train that RFs, and more accurate. We haven't done rigorous experiments to confirm that they do indeed avoid the bias problem. If you try running these experiments, we'd love to hear what you find, and would be happy to help share your findings!

Understanding the reason why extremely randomized trees can help requires understanding why Random Forests are biased. The issue is that each time we select a break point in a variable in a Random Forest, we exhaustively test every level of the variable to find the best break point. This, of course, makes no sense at all, since we're trying to create a semi-randomized tree, so finding the *optimal* split point is a waste of time. Extremely randomized trees avoid this unnecessary step.

As well as being unnecessary, the optimal-split-finding step introduces bias. For a variable with many levels (in the most extreme case, a continuous variable will generally have as many levels as there are rows of data) this means testing many more split points. Testing more split points means there's a higher probability of finding a split that, purely by chance, happens to predict the dependent variable well. Therefore, variables where more splits are tried, will appear more often in the tree. This leads to the bias in the gini importance approach that we found.