Initial Library Imports

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(context='notebook', style='darkgrid')
%config InlineBackend.figure_format = 'retina'

import warnings
warnings.filterwarnings('ignore')
```

Feature Importance and Selection

MSDS689 - Joren Libunao

Introduction

Machine learning models have become a big part of data analysis and decision-making processes in various industries. These models are designed to learn patterns in the data and make predictions based on those patterns. When it comes to building any machine learning model, one of the critical tasks is determining which features or variables to include in the model. These features or variables can be numeric, categorical, or text-based, and they can vary in terms of their importance in predicting the outcome. While you can technically add all features available in the dataset in question, this may cause various issues, including the curse of dimensionality, where having too many features can make it difficult to build accurate models due to increased model complexity.

It is important to consider which features of the dataset you want to include as predictors. This process is known as feature selection, where you essentially simplify your model by only selecting certain features that explain more of the variance in your data than other features. Some features may not contribute much to the prediction and instead only complicate the model unnecessarily. In this report, we will go over various feature selection methods to build an understanding of which methods work best, and some basic feature importance concepts.

- Terminology:
  - Global vs Local Feature Importance
  - Model-Specific vs Model-Agnostic
- Algorithms:
  - Spearman's rank correlation coefficient (regression models)
  - Lasso - L1 Regularization (linear regression models)
  - Permutation importance (tree-based models)
  - Shapley Additive Explanations (SHAP)

Model Evaluation Before Feature Importance

IMPORTANT NOTE: Before conducting any feature importance tests, the very first thing you must do is to evaluate your model performance. If your model performs poorly, then feature importance is meaningless, as you would not even know if the "important" features are accurately contributing to your model or not. If the model is performing well, then analyzing feature importance can help us understand the features' contributions much more clearly. This can also help us further improve the model, explain its behavior, or determine if the model is overfitting.

Terminology

Global vs Local Feature Importance

Before we get into the algorithms, we want to introduce a couple factors about feature importance that are crucial to consider. The first is whether the importance represents global or local importance.

- **Global**: Global feature importance tells us, on average, which features are important across the entire dataset.
- **Local**: Local feature importance tells us the importance of each feature for a specific prediction.

Global feature importance considers all of the samples in a dataset. It can be calculated by aggregating the feature importance values over all of the samples, usually by taking the mean or the sum of the importance scores. This is mainly what we care about when it comes to feature importance, as we want to know which features are globally important, or explain the variance in the data well.

Local feature importance, however, focuses on the importance of each feature for a particular sample or instance in the dataset. It aims to explain how the model arrived at a particular prediction or decision by considering the contribution of each feature for that specific prediction. Common methods for these include Local Interpretable Model-Agnostic Explanations (LIME) and Shapley Additive Explanations (SHAP), the latter of which will be discussed later in this report.

Model-Specific vs Model-Agnostic

Another factor to consider is whether the feature importance method is model-specific or model-agnostic:

- **Model-specific**: Model-specific feature importance methods can only be applied to that specific model.
- **Model-agnostic**: Model-agnostic feature importance methods can be applied to virtually any model.

As the name suggests, model-specific feature importance methods do not generalize to all types of models. Some examples are F-tests and t-tests in linear models like Linear or Logistic Regression, or Gini importance in tree-based models like Random Forests and Gradient Boosting. Model-agnostic feature importance methods, on the other hand, can be applied to any model as they do not depend on model characteristics but rather the features themselves.

However, these methods come with certain issues and cannot be reliably used to rank feature importance.

- In the case of linear models, the t-tests do not order features, making it difficult to compare which features might be more important than others--it only determines whether or not a given feature is statistically significant. F-tests are the same, as they only determine if certain models are statistically significant over other models, but it does not provide a way to rank the importance of individual features.
- In the case of tree-based models, Gini importance may not be a reliable metric as it can become biased towards high-cardinality categorical features, or features with a lot of unique values. These features are much more likely to be selected for splitting, causing bias in the feature importance calculation. We discuss this in the permutation importance section.

Algorithms

Spearman's Rank Correlation Coefficient

One of the simplest ways of calculating feature importance is by using Spearman's rank correlation coefficient, a global, model-agnostic feature importance technique. The coefficient represents the relationship between the rank of each feature and the target variable. If a feature has a high rank correlation coefficient, then its rank is strongly related to the target variable, and if the coefficient is low, then the rank is not strongly related. The method measures single-feature relevance importance and works well for independent features, but will suffer when it comes to codependent features or multicollinearity. The formula for the coefficient is as follows:

$$\rho_s = 1 - \frac{6\Sigma d_i^2}{n(n^2-1)}$$

where

- d_i = difference between the two ranks of each observation
- n = number of observations

To demonstrate, we will use the `spearmanr` implementation in SciPy's `scipy.stats` module to see which features have the highest ranks. We are going to base our analysis on

the diabetes dataset in Scikit-learn.

In [2]:
```
from scipy.stats import spearmanr
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import load_diabetes

data = load_diabetes()
X, y = data.data, data.target
feature_names = data.feature_names

model = RandomForestRegressor(n_estimators=100)
model.fit(X, y)

# calculate Spearman's rank correlation coefficient and p-value for each featu
correlations, p_values = [], []
for i in range(X.shape[1]):
    corr, p = spearmanr(X[:, i], y)
    correlations.append(corr)
    p_values.append(p)

# create a DataFrame to display feature importances
importance_df = pd.DataFrame({'feature': feature_names, 'correlation': correlat
importance_df = importance_df.sort_values('correlation', ascending=False)
importance_df
```

Out[2]:

	feature	correlation	p-value
8	s5	0.589416	1.067821e-42
2	bmi	0.561382	4.567024e-38
7	s4	0.448931	2.629039e-23
3	bp	0.416241	5.992784e-20
9	s6	0.350792	3.034494e-14
4	s1	0.232429	7.791070e-07
0	age	0.197822	2.806132e-05
5	s2	0.195834	3.387255e-05
1	sex	0.037401	4.328319e-01
6	s3	-0.410022	2.377057e-19

The `spearmanr` function allows us to create a summary DataFrame of the features as shown above. The feature `s5` has the highest rank correlation coefficient among all features at 0.589416, indicating a moderately positive correlation between the feature and the target variable. Meanwhile, on the other end, the feature `s3` has the lowest rank correlation coefficient among all features at -0.410022, indicating a moderately negative correlation between it and the target. That doesn't necessarily mean it is unimportant, but rather that it influences the target variable inversely in comparison. This tool can be used to quickly get a sense of which features influence the target variable the most.

Time Complexity - Spearman's Rank Correlation Coefficient

The time complexity of the Spearman's Rank Correlation Coefficient algorithm ultimately depends on the number of samples and number of features in the dataset, but generally the time complexity is $O(n^2)$ or $O(n^3)$ as it must sort the values of each feature and then compare them to the sorted values of the target variable, meaning it must go through all observations twice. For larger datasets, this will obviously be computationally expensive, and so permutation importance may be more feasible.

Lasso - L1 Regularization

Least Absolute Shrinkage and Selection Operator, or Lasso, is a statistical method used for variable selection and regularization in linear regression. The goal of Lasso is to find which input features are most important for output prediction while also preventing overfitting. Lasso achieves this through adding a penalty term to the linear regression loss function (based on Sum of Squares of residuals) like so:

$$\sum_{i=1}^{n}(y_i - \sum_j x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

where λ represents a hyperparameter that must be tuned to control the strength of the L1 regularization penalty term. It is also usually referred to as the amount of shrinkage, as the strength of λ determines which coefficients get reduced to zero. When $\lambda = 0$, no parameters are eliminated and it is essentially a basic linear regression model. But as λ increases, more and more coefficients are set to zero and eliminated. In theory, if $\lambda = \infty$, all coefficients get eliminated. There is also a bit of bias/variance tradeoff, because as λ increases, bias increases, but as λ decreases, variance decreases. As a side note, if there is an intercept term, it is usually not included in the extra Lasso penalty term.

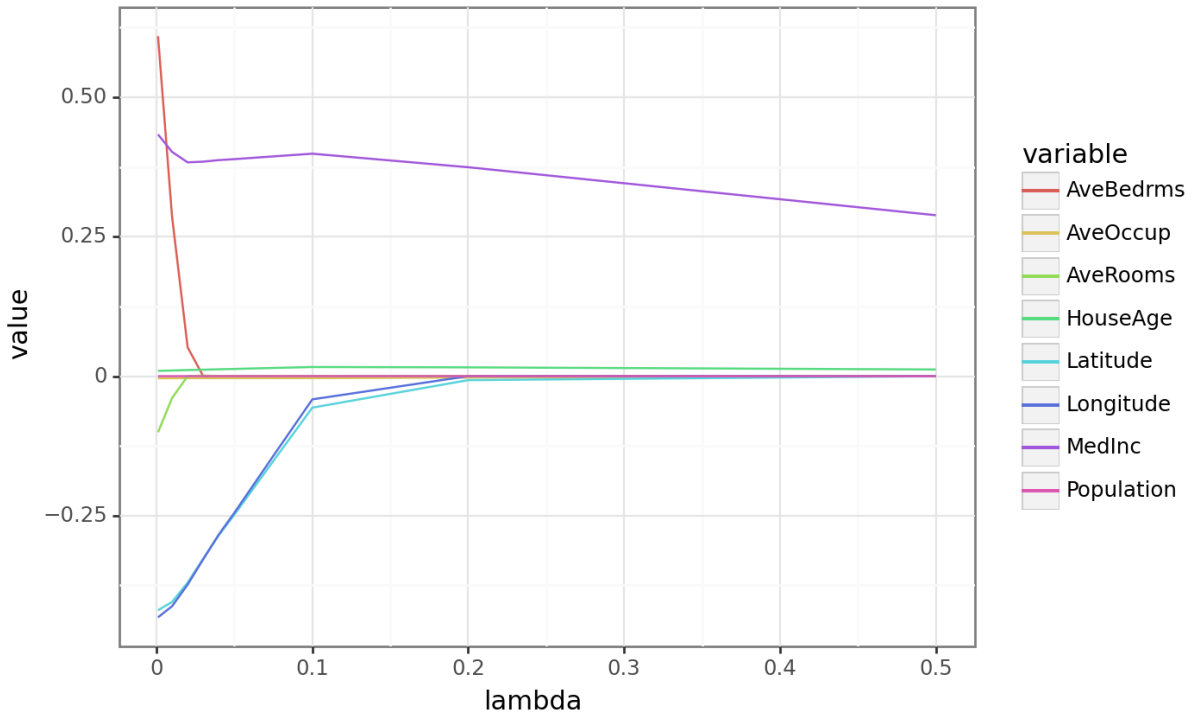
We will look at a brief example of how Lasso can reduce parameters to zero, essentially "selecting" features by leaving the significant coefficients. We will use the California housing dataset in Scikit-learn to test different values of λ to see how they affect coefficient size.

In [3]:
```
from sklearn.datasets import fetch_california_housing
from sklearn.linear_model import Lasso

x, y = fetch_california_housing(as_frame=True, return_X_y=True)

lmbda = [0.001, 0.0025, 0.005, 0.075, 0.01, 0.02, 0.03, 0.04, 0.05, 0.075, 0.1,
betas = np.empty([len(lmbda), x.shape[1]])
for i in range(len(lmbda)):
    lm = Lasso(alpha=lmbda[i], tol=.1)
    lm.fit(x, y)
    betas[i,:] = lm.coef_

coef_df = pd.DataFrame(betas, columns = x.columns)
coef_df['lambda'] = lmbda
coef_df
```

Out[3]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	0.433247	0.009510	-0.100553	0.609329	-0.000004	-0.003773	-0.419624	-0.432271
1	0.428046	0.009622	-0.090353	0.555526	-0.000003	-0.003753	-0.417132	-0.428949
2	0.419398	0.009808	-0.073375	0.465931	-0.000002	-0.003719	-0.412909	-0.423343
3	0.393926	0.014711	-0.000000	0.000000	0.000017	-0.003519	-0.151908	-0.142758
4	0.401990	0.010179	-0.039258	0.286086	-0.000001	-0.003651	-0.404678	-0.412337
5	0.383247	0.010979	0.000000	0.051716	0.000002	-0.003585	-0.370089	-0.373554
6	0.384491	0.011564	0.001051	0.000000	0.000003	-0.003574	-0.327013	-0.328068
7	0.387268	0.012318	0.000000	0.000000	0.000006	-0.003570	-0.284941	-0.283763
8	0.388989	0.012971	0.000000	0.000000	0.000009	-0.003552	-0.248215	-0.244756
9	0.393926	0.014711	-0.000000	0.000000	0.000017	-0.003519	-0.151908	-0.142758
10	0.398747	0.016431	-0.000000	0.000000	0.000024	-0.003484	-0.056399	-0.041557
11	0.374479	0.015815	-0.000000	0.000000	0.000024	-0.002500	-0.007095	-0.000000
12	0.288478	0.011984	0.000000	-0.000000	0.000012	-0.000000	-0.000000	-0.000000

In [4]:
```
from plotnine import ggplot, aes, geom_line, theme_bw

coef_df = coef_df.melt(id_vars = 'lambda')
ggplot(data = coef_df, mapping = aes(x = 'lambda', y = 'value', color = 'variab
```



Out[4]: <ggplot: (8762489292647)>

As you can see above, most of the parameters get zeroed out for $\lambda > 0.2$ with the exception of `MedInc`. Looking at the area between $0 < \lambda < 0.2$, the parameters slowly get zeroed out one by one as λ increases. This demonstrates how Lasso acts as a feature

selection tool by reducing the power of coefficients, and thus their respective predictors, to contribute less to the model prediction.

Time Complexity - Lasso

While it may not technically be an algorithm in the traditional sense, it can behave like other feature selection algorithms as it solves an optimization problem involving minimizing the sum of squared errors plus a constraint that limits the size of coefficients. This problem typically gets solved using gradient descent or other similar algorithms. If it uses gradient descent to optimize, the time complexity of the algorithm is considered $T(knp)$, where k represents the numer of iterations required to converge, n represents the size of the data, and p represents the number of parameters. Asymptotically, this would represent an asymptotic complexity of $O(n)$.

Permutation Importance

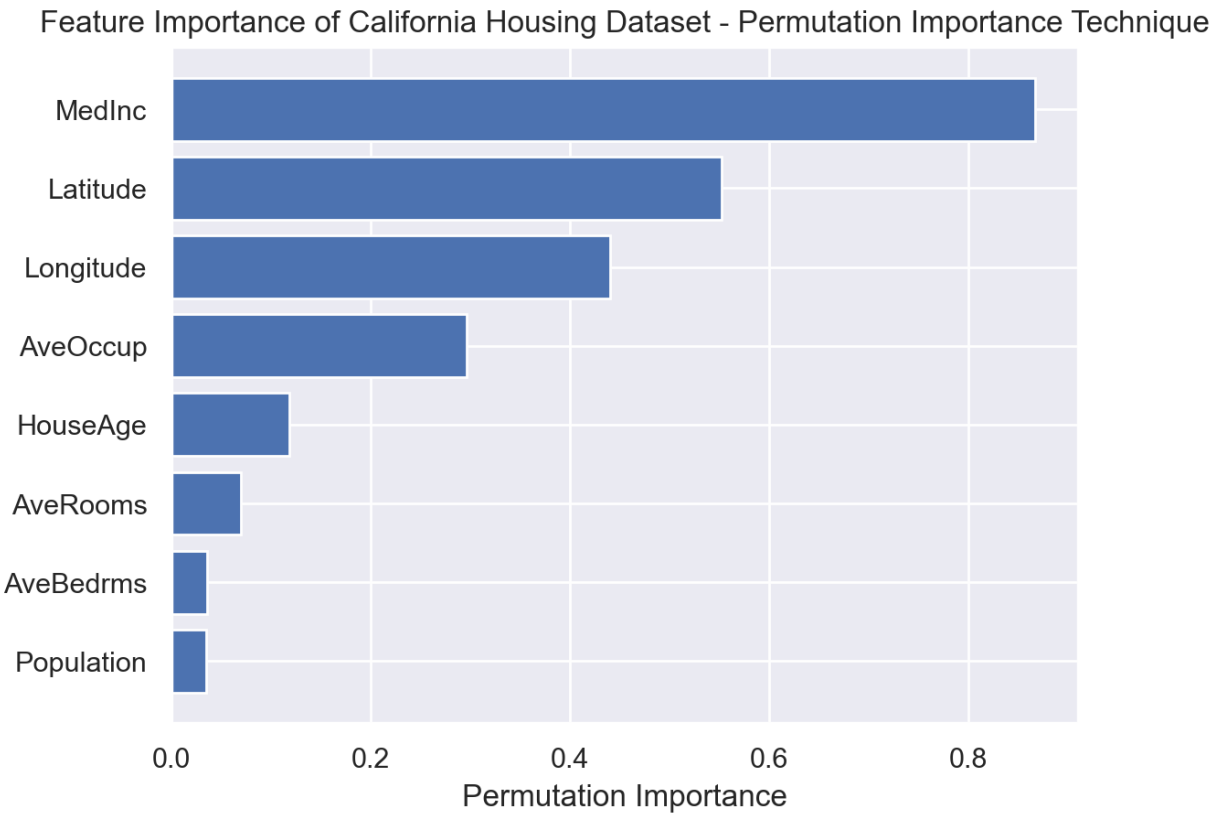
Permutation importance is a global, model-agnostic feature importance technique where you measure the effect of imputing (i.e. shuffling) the data in a given feature on the performance of the model, which is measured by a chosen metric like MSE, R^2, RMSE, etc. The permutation importance algorithm computes this metric on a validation set before and after randomly permuting the values of each feature, and the feature importance is determined by comparing the decrease in the model performance before and after the data shuffle. Features with a large decrease are considered important, suggesting that the model depends on that feature for accurate predictions, while features with little to no effect on model performance are considered insignificant as the model performs similarly with or without the feature's information.

The general algorithm for permutation importance is below:

- **Step 1**: Calculate the total error (s) on some set of data X (i.e. MSE, R^2, accuracy, AUC, etc.)
- For each feature j in X:
  - Repeat K times:
    - **Step 2**: Permute the values in j.
    - **Step 3**: Use new data to get predictions for X.
    - **Step 4**: Compute the new error s_{jk} using data with the permuted feature value.
  - **Step 5**: Calculate feature importance for that given feature j by taking the average of the permuted feature errors and subtracting it from the total error calculated in Step 1:
    $$I_j = s - \frac{1}{K}\sum_{k=1}^{K} s_{jk}$$

Once you have calculated the feature importances for all of your features, you can measure their importance by plotting them. Below, we will demonstrate the `permutation_importance` function from Scikit-learn's `inspection` module by using it on the California housing dataset in Scikit-learn.

In [5]:
```
from sklearn.datasets import fetch_california_housing
from sklearn.ensemble import RandomForestRegressor
from sklearn.inspection import permutation_importance

data = fetch_california_housing(as_frame=True)
X = data.data
y = data.target

model = RandomForestRegressor(n_estimators=100, random_state=0)
model.fit(X, y)

result = permutation_importance(model, X, y, n_repeats=10, random_state=0, n_jo
importance = result.importances_mean
sorted_idx = importance.argsort()

fig, ax = plt.subplots()
ax.barh(range(X.shape[1]), importance[sorted_idx])
ax.set_yticks(range(X.shape[1]))
ax.set_yticklabels(X.columns[sorted_idx])
ax.set_xlabel('Permutation Importance')
ax.set_title('Feature Importance of California Housing Dataset - Permutation Im
plt.show()
```



As you can see in the plot above, `MedInc` has the highest permutation importance with an average of about 0.85, with the next closest being `Latitude` with a permutation importance of roughly 0.55. The lowest permutation importance was `Population` at roughly 0.04.

Permutation importance gives us an easy way to visualize and demonstrate which features are the most important for a given dataset and model, and the output is easily interpretable

as well, making it easy to explain to non-technical peers.

Time Complexity - Permutation Importance

According to online sources, the time complexity of the Scikit-learn implementation of permutation importance is $O(np * log(p))$, where p = the number of features. The algorithm must compute the feature importance scores for each feature by permuting its values and re-evaluating the model, which requires resorting the data as well. Typically, this would be a lot slower, but Scikit-learn's implementation has certain optimizations which speed up the computation, such as parallel processing and cached results.

Permutation Importance in the Context of Random Forests

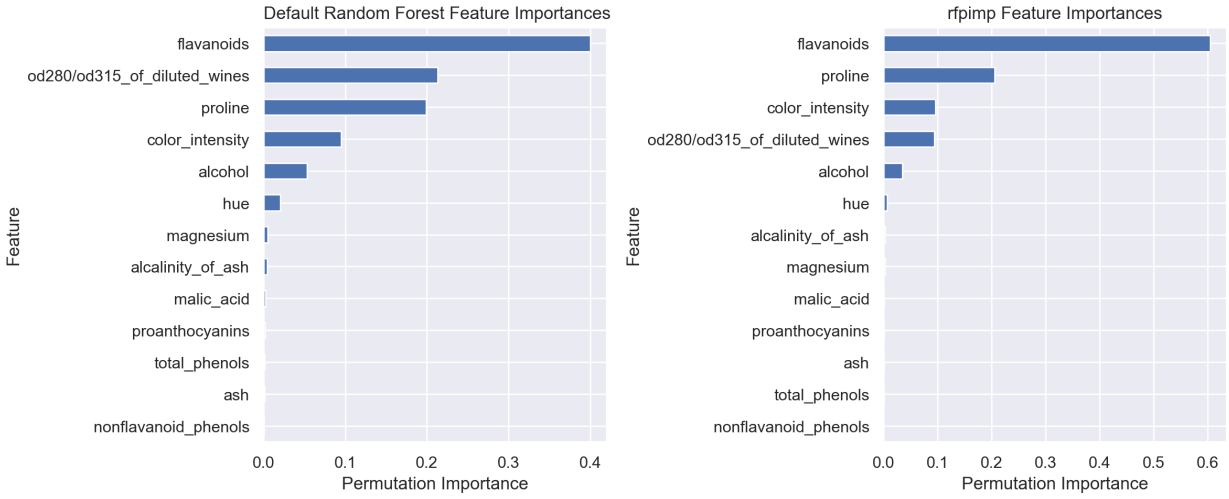
Random Forest models built from Scikit-learn's implementation come with an attribute `feature_importances_` that allow us to easily access the feature importances for the model, but it is not always reliable. The main strategy for calculating feature importance is *mean decrease in impurity*, or Gini importance. As previously mentioned in the Terminology section of this report, Gini importance is unreliable as it can become biased towards categorical features with high cardinality, or a high number of unique categories/values. These features are more likely to be selected for each split, which can cause bias in the feature importance calculation.

To remedy this issue, you can use the `rfpimp` library to get more reliable results. The `rfpimp` library calculates feature importance differently by basing it on the decrease in a model's out-of-bag (OOB) score when a feature is randomly permuted. A quick side note about OOB scores: this value represents the accuracy of the model on examples that were not used in the training of each decision tree in the Random Forest model. This implementation is more useful for understanding feature importance in a Random Forest model as it also takes into account interactions between features and their importance in the overall model. In addition, while it may be called `rfpimp`, it does not necessarily have to be used with a Random Forest model, as it can be extended to any tree-based model, such as Decision Trees by themselves, Gradient Boosting models, AdaBoost models, etc.

Below, we will use the `rfpimp` library to compare feature importance calculations between itself and the Scikit-learn implementation for a model fit on the wine dataset from Scikit-learn.

In [6]:
```
from sklearn.datasets import load_wine
import rfpimp

data = load_wine(as_frame=True)
X, y = data.data, data.target

rf = RandomForestRegressor(random_state=0)
rf.fit(X, y)

fig, ax = plt.subplots(1,2,figsize = (12,5))

feat_importances = pd.Series(rf.feature_importances_, index=X.columns)
```

```
feat_importances.nlargest(13).plot(kind='barh', ax = ax[0]).invert_yaxis()
ax[0].set_title('Default Random Forest Feature Importances', loc='left')
ax[0].set_ylabel('Feature')
ax[0].set_xlabel('Permutation Importance')

# calculate feature importance using rfpimp's "permutation importance" method
perm_imp = rfpimp.importances(rf, X, y)

# plot the comparison
perm_imp.plot(kind='barh', ax=ax[1]).invert_yaxis()
ax[1].set_title('rfpimp Feature Importances')
ax[1].set_xlabel('Permutation Importance')
ax[1].legend().set_visible(False)
fig.tight_layout()
plt.show()
```



As you can see above, the feature importances for each implementation are calculated differently:

- While both may have selected `flavanoids` as the most important feature of the model, the `rfpimp` implementation actually gave it a higher importance than the built-in implementation.
- Secondly, the other less important features were ranked differently and also given different feature importances. The second and third features in the built-in implementation were the `od280/od315_of_diluted_wines` and `proline`, while the `rfpimp` imlementation selected `proline` and `color_intensity` as the second and third most important features, respectively.

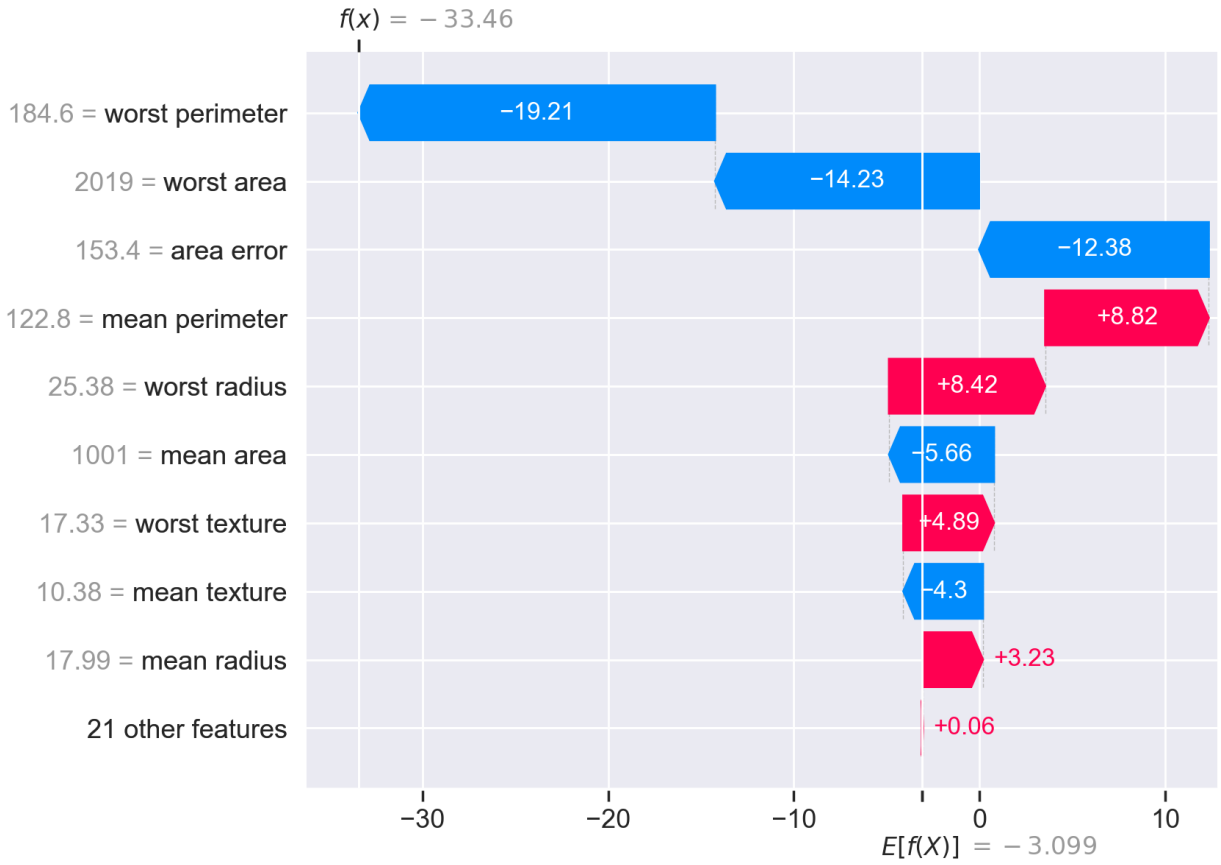
Shapley Additive Explanations (SHAP)

Shapley Additive Explanations, or SHAP, is a local, model-agnostic feature importance approach for interpreting predictions from essentially any machine learning model. It is based on the concept of Shapley values from cooperative game theory, where you assign a value to a feature in a prediction based on its contribution to the overall prediction.

SHAP is a flexible approach to feature importance/selection, and can be applied to both classification, regression, and time series models. The `shap` library in Python provides

methods for plotting your feature importance as well, making it very easy to visualize feature importance with nearly any model. Below, we will use it to visualize feature importance in a Logistic Regression model fit on the breast cancer dataset in Scikit-learn.

In [7]:
```
import shap
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_breast_cancer

data = load_breast_cancer()
X, y = data.data, data.target
feature_names = data.feature_names
df = pd.DataFrame(X, columns=feature_names)
model = LogisticRegression()
model.fit(X, y)

explainer = shap.Explainer(model, df)
shap_values = explainer(df.iloc[:10, :])

shap.plots.waterfall(shap_values[0], max_display=10)
```



This waterfall plot shows the contribution of each feature to the final prediction for a particular instance. The y-axis shows feature names, and the x-axis shows the SHAP value, which represents the amount by which the feature value deviates from its expected value given the other features in the model. A quick summary of the visual elements:

- The length of the bar indicates the magnitude of the contribution of that feature.
- Blue bars indicate features that tend to increase the prediction.
- Features with red bars tend to decrease the prediction.

- The base value (expected output of the model) is shown in gray at the bottom of the plot as `E[f(x)]`, and is "pushed" back and forth by each of the features as you go from bottom to top, with the final prediction value shown at the top as `f(x)`.

Time Complexity - Shapley Additive Explanations

It is difficult to determine a rule of thumb for the time complexity of SHAP, as it depends on the model being explained. With linear models, the time complexity can be linear, or $O(n)$. But for more complex, tree-based models, the time complexity can be much higher, reaching exponential time complexities of $O(k^n)$, where k represents how many subsets of the data SHAP re-evaluates the model on. The reason for this is because SHAP re-evaluates the model with different subsets of features, and so it has to continuously refit the model many times based on however many subsets of features selected.

Conclusion

As we have seen in this report, there are many different ways to measure and check feature importance in machine learning models. Some are specific to the model itself while others are much more generalizable. In addition to those that we explored in this report, there are many other algorithms and techniques that we can use to check feature importance, such as Local Interpretable Model-Agnostic Explanations (LIME), which can be used for various types of data such as image and text data, and is especially useful for black box models where the underlying decision-making process can be difficult to interpret.

Which method to choose ultimately comes down to the context of the situation, i.e. which model is being used, what kind of data is being analyzed, what sort of interpretations are we looking for, etc. But regardless of which method is chosen, a data scientist should always validate their model first, or else their feature importance analysis is absolutely useless.