DATA SCIENCE FROM SCRATCH

Part 2: Business Machine Learning

Can I to I Can

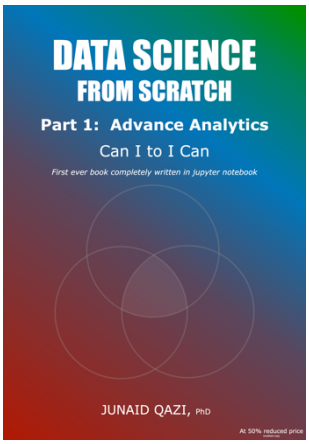
A book completely written in jupyter notebook

JUNAID QAZI, PhD

Preface:

Dear learners,

I am thankful for your interest in the first book which covers all what you need for advance business intelligence. Here is a brief overview on Part 1:

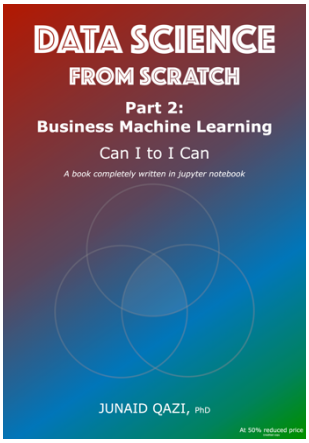


1: **Data Science from Scratch (Part 1: Advance Analytics)**

(1 to 400 pages)

Covers all what you need for advance business intelligence. Everything from scratch to the point where you can learn to implement data preprocess pipeline and presented insights. Most of the time, you only need this part for advance analytics. The idea here is to give you the skills so that you can quickly start looking for projects on freelancing platforms, there is a lot that you can do just after finishing the part 1.

Direct link to the part 1: https://leanpub.com/data-science-from-scratch



The one you are going to read is Part-2 in the series:

2: **Data Science from Scratch (Part 2: Business Machine Learning)**

This part cover all you need to work in business setup to do machine learning. Concepts are introduced when required and the book is rich for extra readings. It's extract of my years of experiences, and I hope you will learn a lot from this book.

Direct link to the part 2: https://leanpub.com/datascience-from-scratch-part-2

This is an early release, and you may find some typo errors. However, all the codes are working and using latest versions of stable libraries till my last check. On page 18, you will find the versions of the libraries that the book is using.

The book is ***completely written in jupyter notebook*** so that, you feel real working environment that is preferred by data science community.

Wishing you good luck for your future plans!

Dr. Junaid S Qazi, PhD
https://www.linkedin.com/in/jqazi/

How to get maximum benefit from this book!

For me, data science is an art and programming play major role in understanding the art of getting insights and building machine learning models. Although, I have included significant theory to make you understand the core concepts, however, the focus is to **learn by doing**.

The best use of this book is, open a jupyter notebook and type in the code yourself. You will see yourself making progress and will understand the ins and outs.

Don't copy past, type yourself and create your own code notebook!

From my personal experience of recruitment data scientists, conducting professional development bootcamps and immersive data science courses, I strongly believe, this book will set strong foundations to pass any data science interview. This book is rich with external links for those who want to explore even more. At several places, I have provided code to get interactive visualization to understand the concepts deeply.

This book covers MUST HAVE supervised machine learning algorithms, frankly speaking, the algorithms in this book are actually solving more than 95% of day-to-day business operational problems. If you truly finish this book, your chances to clear any job interview are very high and it's my **money back guarantee**!

If you have questions, you can always write me!

Good luck!

Dedicated to:

My beloved parents, Yaqoob and Ruqia
My lovely wife, Saba
My beautiful daughter, Zahra
and
My amazing sons, Saad and Sarim
For their continuous support and smiles that they have brought in my life.

Theory Slides – Linear Regression

L1: Linear Regression

If you have not installed scikit-learn so far:

- Anaconda users: **conda install scikit-learn=<version_number>**
- Others: **pip install scikit-learn==<version_number>**

L2: Regularization

L3: Bias-Variance Trade-off

L4: Categorical features

Creating dummies

Theory Slides – Logistic Regression

L5: Logistic Regression

L6: Logistic Regression – Titanic data

Predict if the person was dead or alive!

L7: Logistic Regression – Multiclass Classification

L8: Handling imbalanced classes in the dataset

L9: Predicting Chronic Kidney Disease

Theory Slides – K-nearest neighbors

L10: K-nearest neighbors (KNN) – Working principle hands-on

L11: K Nearest Neighbors – hands-on implementation

L12: Logistic regression vs KNN - breast cancer dataset

Theory Slides – Decision trees and random forests

L13: Decision trees and tree based ensemble learning

L14: Bootstrapping and Confidence Interval

L15: Introduction to SVMs

L16: Support Vector Machines (SVMs) – Hands-on

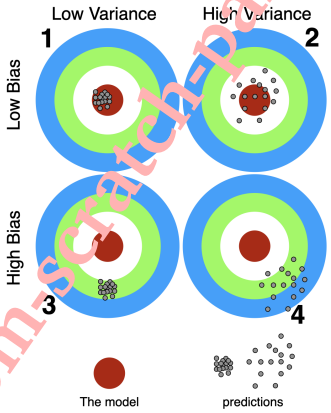
L17: SVMs and Logistic Regression – practice and comparisons

****
****

Bias and Variance

- If we have good distribution in our training data, the model predicts very well and close to the bulls-eye.
- If our training data is full of outliers or non-standard values, this results in poorer predictions.
- These different realizations result in a scatter of hits on the target.

Here comes Bias-variance tradeoff



****
****

Bias Variance Tradeoff

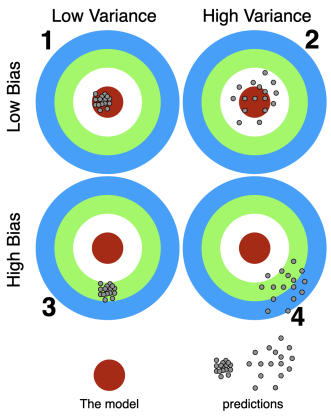
Still wondering about the numbers from 1 to 4 on the diagram, let's look at them. The should make sense now!

1. Low Bias - Low Variance:
- Predicts correct values on the bulls-eye

2. Low Bias - High Variance:
- Predicts values around the bulls-eye with high degree of variance

3. High Bias - Low Variance:
- Predictions would be high bias at a certain location with low variance.

4. High Bias - High Variance:
- Predictions are all over the places



****
****

Linear Regression

(Theory and hands-on)

– End-to-end project using scikit-learn –

Author: Dr. Junaid Qazi, PhD

```
pickle.dump(obj=scaler, file=open(file='transformation.pkl', mode='wb')) # Saving␣
 ↪the transformation
scaler = pickle.load(file=open(file='transformation.pkl', mode='rb')) # Loading␣
 ↪saved transformation
X_scaled = scaler.transform(X) # transforming features
```

[21]:
```
# check the difference!
X_scaled=pd.DataFrame(X_scaled,columns=X.columns) #just creating a dataframe for␣
 ↪scaled features
X_scaled.head(2)
```

[21]:

	CRIM	RM	LSTAT	DIS	NOX
0	-0.419782	0.413672	-1.075562	0.140214	-0.144217
1	-0.417339	0.194274	-0.492439	0.557160	-0.740262

Save the transformation - A good practice

In the above cell, we have standardized all the features before splitting them in trained and test data set. It is important to know that the model trained on standardized features, needs standardized unseen features to make predictions, hence it is recommended and considered a good practice to serialize/save the transformation from training dataset. We can then load it and transform the unseen data before making predictions. Don't worry, we will do this whole process in KNN lecture, wait till that! Try to understand the code below in the meantime!

Go to: L1: Linear Regression

(L1-4-2) Linear Regression Model Training

Excited!

Time to train our very first Machine Learning model!

Train Test Split

Now, we have features in $\boldsymbol{X}$ and target (price) in $\boldsymbol{y}$.

Next step is to split the data into:

- a training set (`X_train` & `y_train`) and
- a testing set (`X_test` & `y_test`).

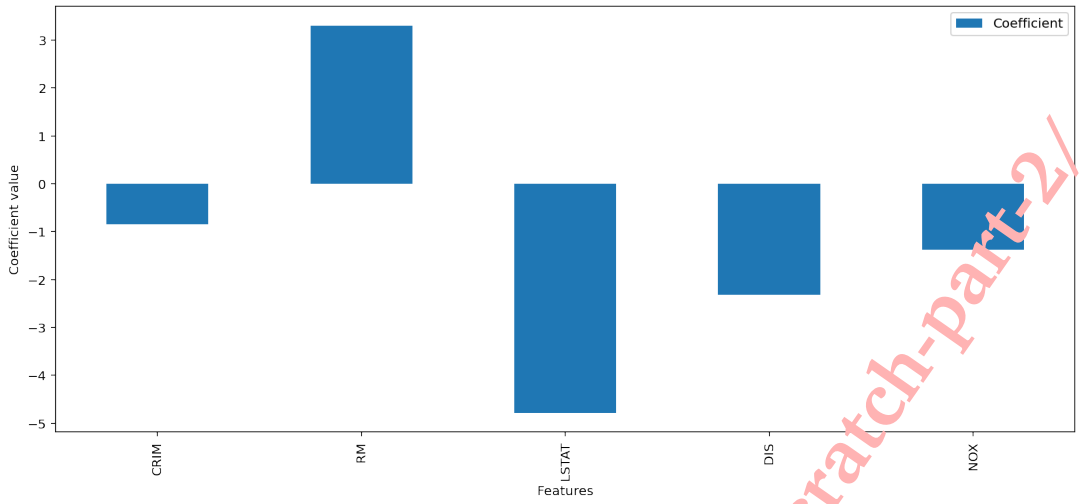
This splitting is important, and can be conveniently done using `Scikit-learn` built-in method `train_test_split`. After splitting, we will train our model on training part of the dataset, which is in `X_train` & `y_train` and then use `X_test` from the test part of our dataset to get the predictions from our trained model. `X_test` will serve as unknown data to the trained model. As we already know `y_test`, that are the actual target values of `X_test`, now comparing model `predictions` and `y_test` will provide help us to evaluate the model performance. *Confusing, don't worry, you will get a clear picture in a while.*

Let's import `train_test_split` method and pass in data along with the `test_size`, which is the % of the dataset that we want in the test part of our data.

> Do you want to save some typing, after import, `<Shift+Tab>` and copy paste `train_test_split` from the end of DocString!

Before we move forward, it is important to briefly discuss these three parameters:

- `test_size` and `train_size`: If we don't pass any value for both of these parameters, `test_size` will set to 0.25 (25% data will go to test). However, if any of these parameter is given some value between 0 and 1, the other will set to the complement of this given value. Remember, the sum of both of `test_size` and `train_size` should not be greater than 1.
- `random_state`, default is `None` and uses the global random state instance from `numpy.random`. Calling the function multiple times will reuse the same instance, and will produce different results



The histogram looks good, however, it is important to know how to explain the values of our model coefficients.

- **Starting with** `RM`, which got the biggest value (~ 3.3...), which suggest that if we keep all other coefficients constant, a one unit increase in the RM is associated with an increase of 3.3... in the price.
- The same is for other related coefficients. e.g. `NOX = -1.37.....` (`Nitric Oxide Concentration`), `DIS = -2.3....` (`weighted distances to five Boston employment centers`) etc decreases the price according to their coefficients, keeping all other constants.
- `LSTAT` - (% lower status of the population), got the biggest negative value of its coefficient (`-4.7....`), which means, it has the highest effect to reduce the house price, however, surprisingly, `CRIM` - Crime Rate, is the least concern of the buyer, it is not contributing significantly to reduce the price of the house in Boston area.

==> *Please note, the coefficients values will be changed if we use full dataset will all features and/or different set of training data, however, the explanation for resulted -ve and +ve features will not change.*

If you want further detail and mathematics behind these explanations, please read the suggested reading assignments!

Go to: L1: Linear Regression

(L1-4-4) Predictions from Trained Model

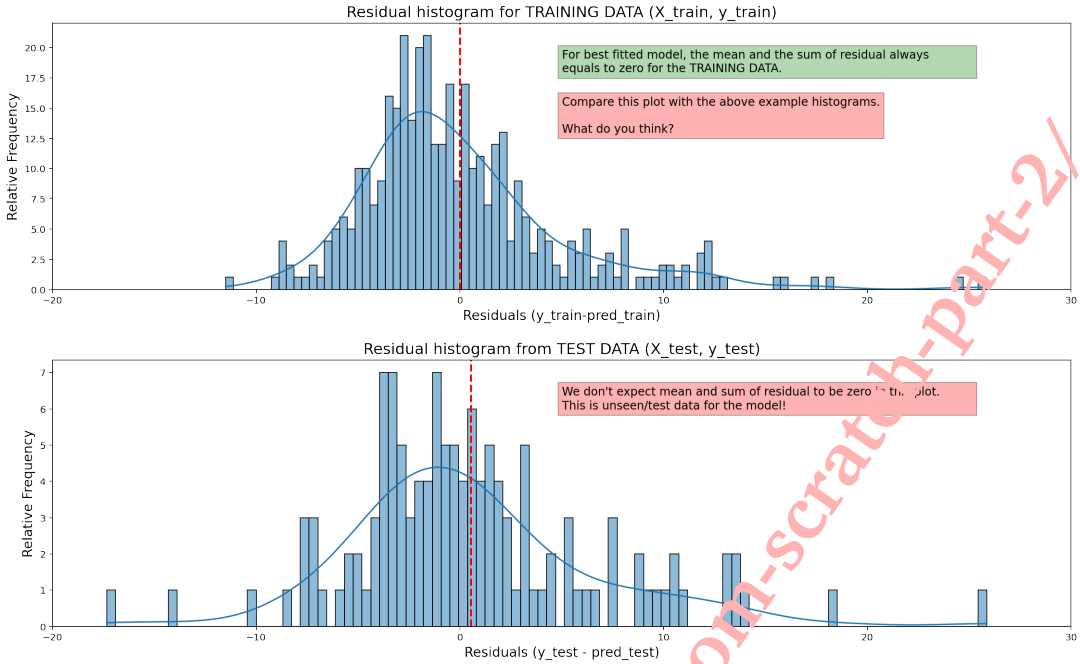
We have trained our model, discussed the coefficients which make some sense, now, its important to know how well the model is doing!

Our model have never seen `X_test`, let's provide test data "`X_test`" to our created model and see what the predictions are. Once we get predictions from the model, we can compare them with what we have in our `y_test` (*known values, its supervised learning!*).

[32]:
```
# Getting predictions from trained model
predictions = lm.predict(X_test)
#predictions # in case you want to see the numbers!
```

We already know the price of all homes with features in `X_test`, which is in `y_test`, let's plot `y_test` and predictions, scatter plot is a good option!. You can compare the true and predicted values.

[33]:
```
plt.figure(figsize=(14, 6))
plt.scatter(x=y_test,y=predictions)
```



Well, we have trained our model "lm". The residual plot does not look bad!

Go to: L1: Linear Regression

Let's see what is the accuracy of our model. We can call score function on our trained model for this purpose, or we can use r2_score function from sklearn.metrics.

Let's try both!

(L1-5-2) R-Square and Goodness of The Fit

(Accuracy Score – R^2)

[37]:
```
# Calling score on trained model "lm"
print("R^2 (train) - The accuracy score of our model in train part is: ", lm.
 ↪score(X = X_train, y = y_train))
print("R^2 (test)  - The accuracy score of our model on test part is: ", lm.score(X␣
 ↪= X_test, y = y_test))

# In case, you don't want long number, try round function - code below
# print("The accuracy score of our model is: ",round(lm.score(X = X_test, y =␣
 ↪y_test), 2))
```

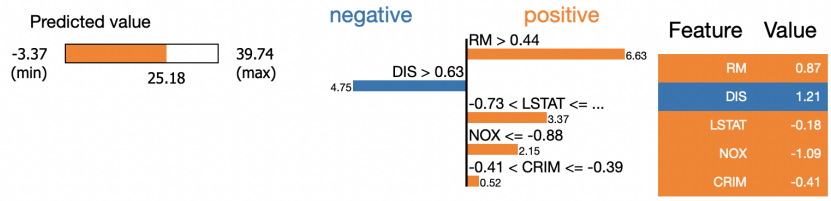
```
R^2 (train) - The accuracy score of our model in train part is:
0.6741099619947835
R^2 (test)  - The accuracy score of our model on test part is:
0.6413655948972747
```

R^2_{adj} is always preferred for multi-variant (multiple) linear regression, the one where we have more than one predictor variables.

[38]:
```
# Adjusted R^2 (see explanation "R^2 vs R^2_adj" at the end)
r_2_train = lm.score(X = X_train, y = y_train)
r_2_test  = lm.score(X = X_test, y = y_test)
```

```
the selected data point.
```

```
<IPython.core.display.HTML object>
```



Each feature's contribution to this prediction is shown in the bar plot (middle). The table on right shows the passed data point whereas, orange color signifies the positive impact and blue signifies the negative impact of that feature on the target. For example, `RM` has a highest positive impact on house price for this predicted value whereas, `DIS` has the highest negative effect on the prediction.

Want to know more, please read the last part What are explanations?

Go to: L1: Linear Regression

(L1-6-2) Model Explanation using SHAP – SHapley-Additive exPlanations

[44]:
```
# Install the library if you don't have
#!pip install shap
```

[45]:
```
# Importing and initialization java script for visualization from SHAP, we can use␣
 ↪matplotlib as well!
import shap
shap.initjs() # jave script
```

```
<IPython.core.display.HTML object>
```

[46]:
```
# Initializing explainer from shap, lm is our trained model
explainer_shap = shap.Explainer(model=lm, masker=X_train) #<shift+tab> for DocString

# Computing shap values for train dataset
shap_values = explainer_shap.shap_values(X_train)
```

Once, we have shap values, we can get force plot for any observation, let's try for the same observation as from LIME.

[47]:
```
# The base value, reference value that the feature contributions start from.
print("The base value is: ", explainer_shap.expected_value)
```

```
The base value is:  22.28486538204206
```

[48]:
```
# Explaining the ith (see LIME part for the value of i=10) observation.
# Visualize the ith prediction's explanation (use matplotlib=True to avoid␣
 ↪Javascript)
shap.force_plot(base_value=explainer_shap.expected_value,
                shap_values=shap_values[i,:],
                features=X_train.iloc[i,:])#,
                #matplotlib=True)
```

```
                                        value = LSTAT, min = 0, max = 100, step = 0.
↪2,
                                        style = {'description_width': 'initial'})
        # Distance
        self.DIS = widgets.BoundedFloatText(description = 'Distance: ',
                                        value = DIS, min = 0, max = 1000, step = 0.1,
                                        style = {'description_width': 'initial'})
        # Nitric oxide concentration
        self.NOX = widgets.BoundedFloatText(description = 'Nitric oxide␣
↪concentration: ',
                                        value = NOX, min = 0, max = 100, step = 0.05,
                                        style = {'description_width': 'initial'})
        display(self.CRIM, self.RM, self.LSTAT, self.DIS, self.NOX)

print("\nEnter observed features to get estimate of the house price:\n")
get_input = input_house_features() #get_input is an instance of our class␣
↪"input_house_features"
```

Enter observed features to get estimate of the house price:

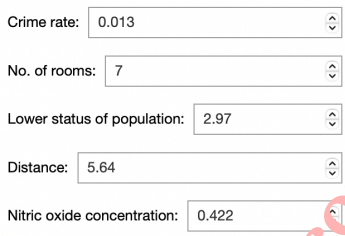
A Jupyter Widget

A Jupyter Widget

A Jupyter Widget

A Jupyter Widget

A Jupyter Widget



[59]:
```
# load the model from disk
filename = 'final_model.sav' # already
model = pickle.load(open(filename, 'rb')) # rb stands for reading only in binary␣
↪format
X_observed={"CRIM":[get_input.CRIM.value], "RM":[get_input.RM.value],
           "LSTAT":[get_input.LSTAT.value], "DIS":[get_input.DIS.value], "NOX":
↪[get_input.NOX.value]}
print("\nThe observed features of the house: \n", X_observed)
print("\nLoading saved transfromation for feature scaling.")
scaler = pickle.load(file=open(file='transformation.pkl', mode='rb'))#loading␣
↪transformation
print("Scaling the given features.")
X_observed_scaled=scaler.transform(pd.DataFrame(X_observed))
print("\nThe observed featrues of the house after scaling transformation:")
print("CRIM: {}, RM: {}, LSTAT: {}, DIS: {}, NOX: {}".
↪format(round(X_observed_scaled[0][0],3),

↪round(X_observed_scaled[0][1],3),
```

```
                                                                    ␣
 ↪round(X_observed_scaled[0][2],3),
                                                                    ␣
 ↪round(X_observed_scaled[0][3],3),
                                                                    ␣
 ↪round(X_observed_scaled[0][4],3)))
predicted_price = round(model.predict(X_observed_scaled)[0],2)
if predicted_price < 0.0: # We don't want to sell in -ve price (lower than 0)!
    print("\nSorry, This house is not in sellable conditions.")
else:
    print("\nEstimated house price, based on the observed features is: {}␣
 ↪Millions\n".format(predicted_price))
```

```
The observed features of the house:
 {'CRIM': [1.513], 'RM': [10.0], 'LSTAT': [35.97], 'DIS': [25.64], 'NOX':
[0.022]}

Loading saved transfromation for feature scaling.
Scaling the given features.

The observed featrues of the house after scaling transformation:
CRIM: -0.244, RM: 5.293, LSTAT: 3.268, DIS: 10.384, NOX: -4.602

Estimated house price, based on the observed features is: 0.73 Millions
```

Go to: L1: Linear Regression

(L1-9) Cross Validation

(SELF STUDY) – Now, I want to introduce another very important concept of Cross Validation at this stage. You know what, my aim is to introduce the thing at the stage where you need them so that you can remember with some context.

These notes are written in a way that we can use them as a reference!

Recall, we have already learned about **overfitting** and **underfitting** along with **bias variance trade-off** in the theory lecture. We are always looking for a sweet-spot between over and under fitting (recall the plot from theory lecture).

We have used train/test split in the above model, where we simply divided our data into `train (X_train, y_train)` and `test (X_test, y_test)` datasets with some percentage. We trained our regression model on the training part and tested/validate on the test part. Both train/test split and cross validation help to avoid overfitting more than underfitting, however, train/test split does have its dangers:

- What if the split we make is not random?
- What if one subset (train/test) of our data has only one type of datapoints and is not a true representative of out complete dataset *(in a simplest case, you can consider your data to be ordered by number of rooms and you get only the rooms with more number is test data)*.

This will result in overfitting, and we don't want this. **This is where cross validation plays it's role**.

Let's move on and learn about the cross validation now. It is a very simple concept and somehow similar to train/test split.

The figure below is taken from scikit-learn's official documentation - fair use policy

Regularization

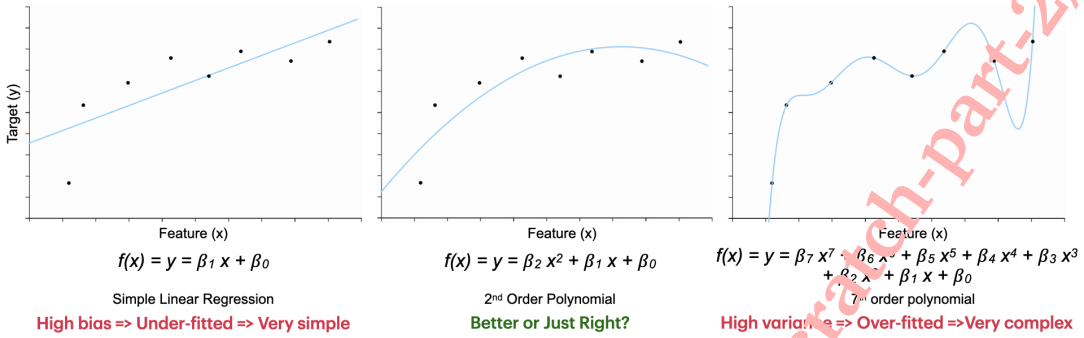
(Ridge, Lasso, Elastic-Net – theory and hands-on)

– Visualize the effects and find the best alpha! –

Author: Dr. Junaid Qazi, PhD

(L2-1) Under-fitting vs Over-fitting

Let's start with these plots to understand under-fitted and over-fitted models.



Go to: L2: Regularization

- **Left plot:** Under-fitted model with low accuracy score (R-square) and higher error (SSE: Sum of Squared Error)
- **Middle plot:** Model with moderate accuracy score and error. Can this be improved?
- **Right plot:** Over-fitted model with very high accuracy score and low error (practically R-square = 1 and SSE = 0 in this case as the fitted line is passing through all the data points).

The plot in the middle and right are fitted with polynomial regression for a single predictor/feature(x). These models are non-linear in the feature(x) space, but linear in the parameters, β, space. Although these models allow for a nonlinear relationship between the target(y), and the feature(x), ***polynomial regression is still considered linear regression since it is linear in the regression coefficients,*** $\beta_1, \beta_2, \ldots, \beta_n$ ***!***

Coefficients β *are also represented by weights within literature. Typically, a regression carried out on standardized variables produces standardized (regression) coefficients,* β^*.

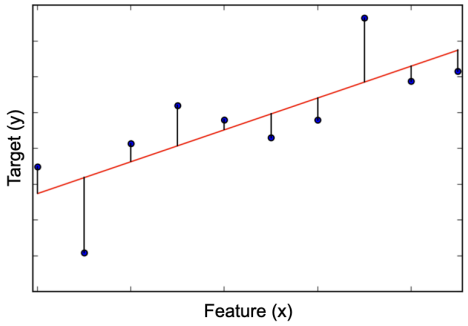
Finding the optimal regression model for generalization is a very challenging task! This is where regularization can play its role to improve generalizability of the trained model.

Please note: The Residual Sum of Squares (RSS), also known as the Sum of Squared Residuals (SSR) or the Sum of Squared estimate of Errors (SSE)

(L2-2) The Least Squares Loss Function – Residual Sum of Squares

(Loss functions are also known as cost or objective functions)

Let's have a quick look at out typical loss function first!

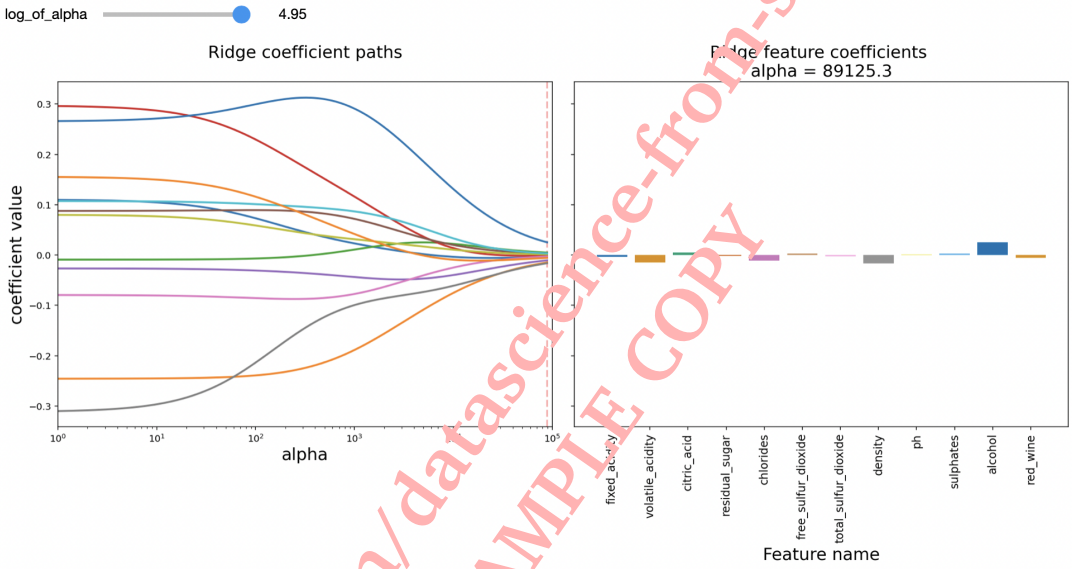


```
from IPython.display import display
```

The function and `interact` from ipywidgets lets me take some specified alphas that we have already calculated the coefficients for and plot them out.

[22]:
```
def visualize_ridge_plots(log_of_alpha=0):
    coefficients_plotter(alphas=r_alphas, coefs=r_coefs, feature_names=features.
 ↪columns,
                         upperbound_alpha=10**log_of_alpha, regtype='Ridge')

interact(visualize_ridge_plots, log_of_alpha=(0.000001,5.0,0.05));
# Remember, we are not computing anything here, our alphas are in r_alphas...
# log_of_alpha is just for plotting, explore the coefficients_plotter function for␣
 ↪details.
```

```
A Jupyter Widget
```



Go to: L2: Regularization

(L2-7-4) The Lasso Effects – Visualizations

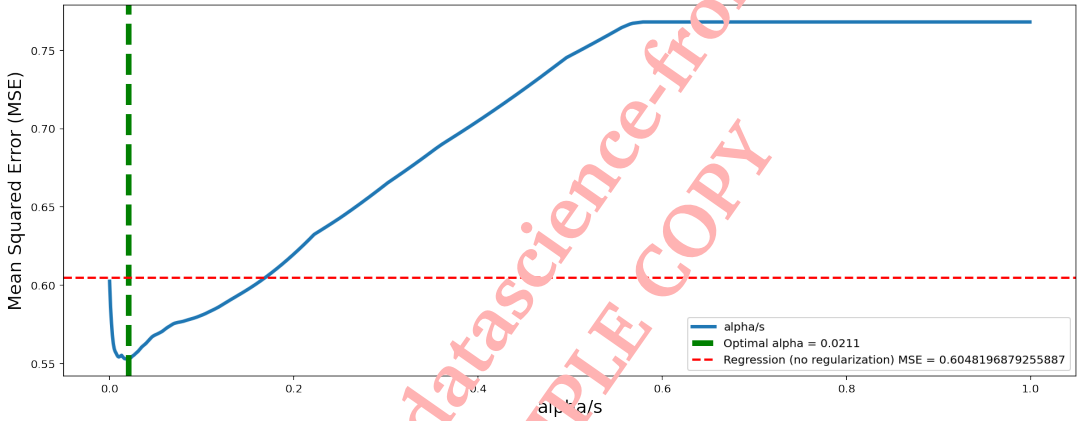
Let's import `lasso` module and see its regularization effects.

[23]:
```
from sklearn.linear_model import Lasso
```

[24]:
```
# This is the same as the ridge_coefficients, but for lasso here to return the␣
 ↪lasso coefficients.
# if we want, we can write a single function for all three -- Ridge, Lasso and␣
 ↪Elastic-Net!
def lasso_coefs(X, y, alphas):
    coefs = []
    lasso_reg = Lasso()

    for a in alphas:
        lasso_reg.set_params(alpha=a)
        lasso_reg.fit(X, y)
```

```
enet_cv_means = [np.mean(cv_alpha) for cv_alpha in enet_model.mse_path_]

plot_cv(enet_model.alphas_, enet_cv_means, enet_optimal_alpha, lr_cv_mean_mse)

print("Computing mean R^2 for the best found value of alpha -- Lasso")
print("R^2 is {} for alpha best {}".format(-cross_val_score(
    estimator=ElasticNet(alpha=enet_optimal_alpha),
    X=X_overfit, y=y_overfit, cv=5,
    scoring='neg_mean_squared_error').mean(), enet_optimal_alpha ))
```

```
importing: from sklearn.linear_model import ElasticNet, ElasticNetCV
Generating values for alpha for lasso...
We will test 1000 alpha/s with range (0.0001,0.9991)...
Elastic-Net ratio 'l1_ratio' = 0.7
Optimizing Elastic-Net model on the given dataset...
Finding best alpha...
The best alpha for ElastNet is 0.0211
Computing mean R^2 for the best found value of alpha -- Lasso
R^2 is 0.5552914793730197 for alpha best 0.0211
```



Now, the questions are:

- Which model you prefer for your `(X_overfit, y_overfit)` dataset?
- Can you find a better model for this dataset?

Try yourself!

Go to: L2: Regularization

Good luck!

Refresher for you!

- Regression with Matrix Algebra
- A Matrix Formulation of the Multiple Regression Model

Bias-Variance Trade-off

(Oversimplified vs complex – theory and hands-on)

– Finding the sweat-spot! –

Author: Dr. Junaid Qazi, PhD

1. **Irreducible Error** from an **imperfect ability to measure** morale because of some unavoidable reasons.
2. **Bias Error** from an **imperfect relationship** between time and morale.
3. **Variance Error** from an **insufficient amount of GOOD data** that can correctly quantify the relationship/s.

All of these sources of errors combine together resulting into the final error in our trained model.

Remember, We always have error in our models, however, it depends how much and what proportion of each type. We can play with bias and variance to find the sweet-spot, however, we can't do anything for the irreducible error!

Go to: L3: Bias-Variance Trade-off

(L3-4) Model Error – The Total Error

Having said, there are three sources of error in a model

$$\text{Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

We merely try to pin down where these different contributions are coming from in our model's error and look for the average value that we expect to observe for the error (MSE) measured across all samples and all data points that are given to a particular model for training.

If you to want know little more on the above relationship, here is the typical formula, along with explanation on each component:

$$E\Big[\big(y - \hat{f}(x)\big)^2\Big] = \big(\, \mathrm{E}[\, \hat{f}(x)\,] - f(x)\,\big)^2 + \mathrm{E}\big[\, \big(\, \hat{f}(x) - \mathrm{E}[\, \hat{f}(x)\,]\big)^2\,\big] + E\Big[\big(y - f(x)\big)^2\Big]$$

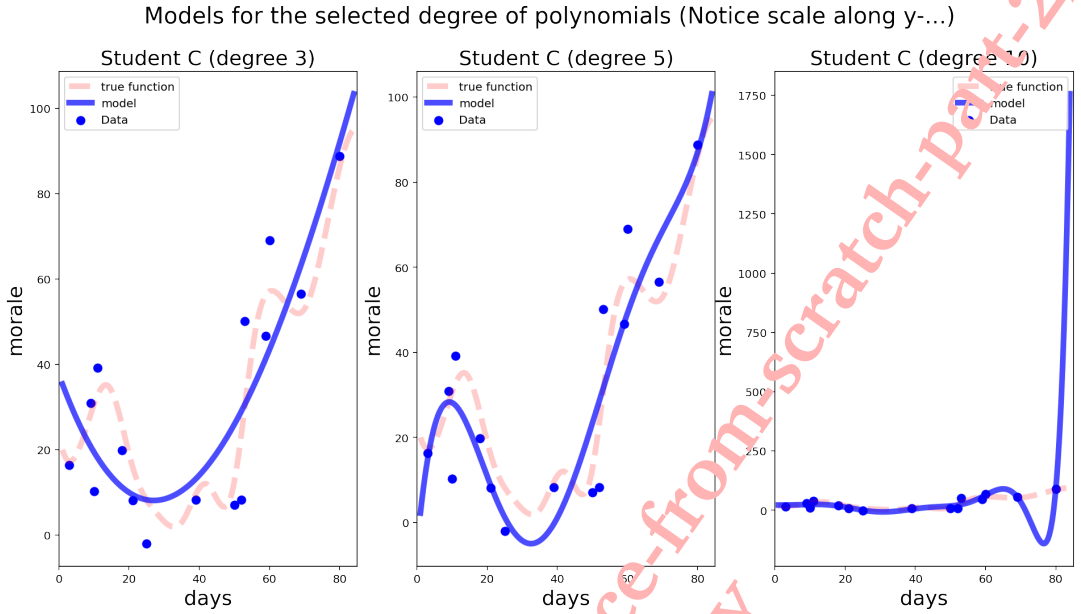
In the above equation:

- $f(x)$ is the true function of y given the features/predictors.
- $\hat{f}(x)$ is the estimate of y with the model fit on a random sample of the predictors.
- $E\Big[\big(y - \hat{f}(x)\big)^2\Big]$ is the mean squared error across multiple models fit on different random samples between the model and the true function.
- $\mathrm{E}[\, \hat{f}(x)\,]$ is the average of estimates for given predictors across multiple models fit on different random samples.
- $E\Big[\big(y - f(x)\big)^2\Big]$ is the mean squared error between the true values and the predictions from the true function of the predictors. This is the **irreducible error**.
- $\big(\, \mathrm{E}[\, \hat{f}(x)\,] - f(x)\,\big)^2$ is the squared error between the average predictions across multiple models fit on different random samples and the prediction of the true function. This is the **bias** (squared).
- $\mathrm{E}\big[\, \big(\, \hat{f}(x) - \mathrm{E}[\, \hat{f}(x)\,]\big)^2\,\big]$ is the average squared difference between individual model predictions and the average prediction of models across multiple random samples. This is the **variance**.

The irreducible error is "noise" – error in the measurement of our target that cannot be accounted for by our predictors.

- The **true function represents the most perfect relationship** between predictors and target, but that does not mean that our variables can perfectly predict the target.
- The **irreducible error can be thought of as the measurement error**: variation in the target that we cannot represent.

[20]:
```
# for student C
plot_polynomial_fits(student_C_days, student_C_morale, morale_func,
             degrees=[3,5,10],student_color='blue', student_name='C')
```



The above plots looks good — ***complex our model is, more variance the model is capturing.....!*** — **Remember:** Increasing the complexity of the model to capture more variance at the expense of reliable and good predictions returns "over-fitted" model.

Go to: L3: Bias-Variance Trade-off

(L3-10-2) Complexity and Simplicity – High Variance and High Bias

It is a struggle to find out the right trade-off between bias & variance, as we are trying to simultaneously minimize these two sources of error that prevent supervised learning algorithms from generalizing beyond their training dataset.

High variance and over-fitting are intrinsically related; if the model predictions are inconsistent across samples, the model is more likely to make the wrong predictions on unseen data in future.

High bias and under-fitting are related; If the model is too basic, it may give very consistent predictions but at the cost of oversimplifying the relationship between the target and predictors.

Go to: L3: Bias-Variance Trade-off

Working with Categorical Features

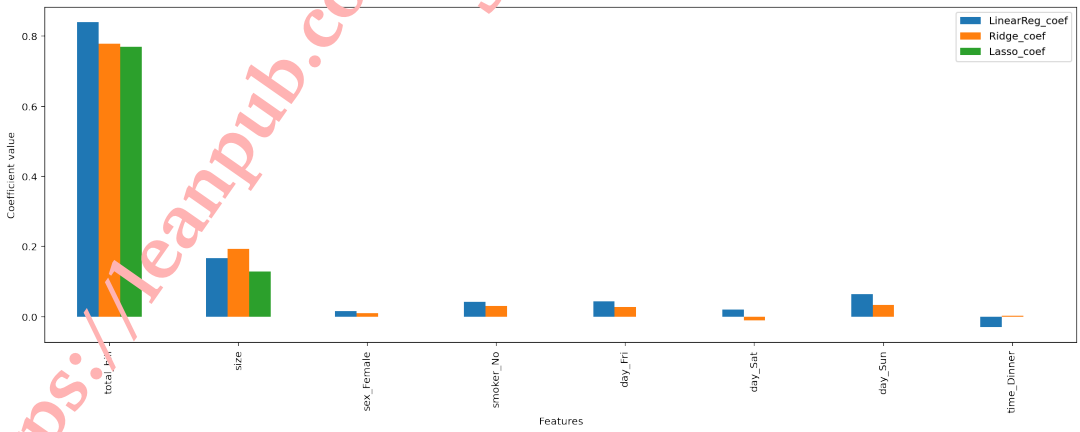
(Creating dummies – categorical features)

– Theory and hands-on –

Author: Dr. Junaid Qazi, PhD

```
lasso_alphas_ = np.arange(0.001, 0.15, 0.001)
lasso_model = LassoCV(alphas=lasso_alphas_, cv=5)
lasso_model = lasso_model.fit(X, y)
lasso_optimal_alpha = lasso_model.alpha_

# Best alpha values and score
# no alpha for linear regression, we know this!
print("ridge_optimal_alpha =", ridge_optimal_alpha)
print("lasso_optimal_alpha =", lasso_optimal_alpha)
print()
print("linear reg. score (R^2) =", lr_model.score(X, y))
print("ridge  reg. score (R^2) =", ridge_model.score(X, y))
print("lasso  reg. score (R^2) =", lasso_model.score(X, y))

# getting all the coefficients of our trained models in a dataframe...!
coeffs = pd.DataFrame(data=lr_model.coef_, index=X.columns,␣
 ↪columns=['LinearReg_coef'])
coeffs['Ridge_coef'] = ridge_model.coef_
coeffs['Lasso_coef'] = lasso_model.coef_
#coeffs # this is the coefficients dataframe

# Using panda's built-in visualization, good to revise!
coeffs.plot(kind='bar', figsize = (18,6))
plt.xlabel('Features'); plt.ylabel('Coefficient value'); # two statements in one␣
 ↪line using ";"
print()
print("Running time: {} sec".format(round(time.time()-start,2)))
```

```
ridge_optimal_alpha = 13.216641839466051
lasso_optimal_alpha = 0.0860000000000001

linear reg. score (R^2) = 0.4700781232060794
ridge  reg. score (R^2) = 0.4688177189564651
lasso  reg. score (R^2) = 0.4630160273207399

Running time: 10.05 sec
```



We can see Lasso reduces coefficients for many column to zero, thinking that they are not important. In case, we have all the redundant columns, they will also get zero coefficients by lasso. As discussed above, it is much better to avoid having redundant variables at the first place, if we keep the redundant columns in the data, we will not have much control which one is driven to zero and it may end up

****
****

Logistic Regression

(Model Evaluation)

Confusion matrix:

Let's define the basic terminology:

n = 100	Predicted No	Predicted Yes
Actual No = 43	TN = 40	FP = 3
Actual Yes = 57	FN = 7	TP = 50
	47	53

- **True Negatives (TN):** Our model predicted No, and they don't have the disease (correct).
- **True Positives (TP):** Our model predicted Yes, and they do have the disease (correct).
- **False Positives (FP):** Our model predicted Yes, but they don't actually have the disease (wrong prediction - known as a "Type I error").
- **False Negatives (FN):** Our model predicted No, but they actually do have the disease (wrong prediction - known as a "Type II error").

****
****

Logistic Regression

(Model Evaluation)

Confusion matrix:

n = 100	Predicted No	Predicted Yes
Actual No = 43	TN = 40	FP = 3
Actual Yes = 57	FN = 7	TP = 50
	47	53

Accuracy: Overall, how often our model predicted correct?

Accuracy = correct predictions / total

Accuracy = (TN + TP) / total = 90 / 100 = **0.90**

Misclassification Rate / Error Rate: Overall, how often our model predicted wrong?

Error Rate = wrong predictions / total

Error Rate = (FP + FN) / total = 10 / 100 = **0.10**

Specificity: When it's actually No, how often does the model predicts No?

Specificity = TN / actual No = 40 / 43 = **0.93**

Precision: When it predicts yes, how often the model is correct?

Precision = TP / predicted Yes = 50 / 53 = **0.94**

****
****

Logistic Regression

(Theory and hands-on)

– All you need to know for model implementation and its evaluation –

Author: Dr. Junaid Qazi, PhD

[1]:
```
# We are already familiar with these libraries!
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# scikit-learn imports
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.preprocessing import StandardScaler

#Setting display format to retina in matplotlib to see better quality images.
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('retina')
#%config InlineBackend.figure_format = 'retina' # this will also work!

# (Optional - good to know)
# Setting 4 digits of precision for floating point output!
#np.set_printoptions(precision=4)

from scipy import stats

# Lines below are just to ignore warnings
import warnings
warnings.filterwarnings('ignore')
```

(L5-1) Probability and Odds

Before we move on on to work with logistic regression, we must have clear understanding for these very important statistical concepts.

Probability

Probability is describing the likeliness of some event to happen or occur on a numerical scale between `0 (impossible) & 1 (certain)`. The higher the probability is, the more likely the event will occur.

> Tossing a fair coin or rolling a dice and expecting how often we will get a head and a certain number on a dice, its simply the outcome divided by the total options or possibilities.

$$Probability = \frac{One\ outcome\ /\ event}{All\ possible\ outcomes\ /\ events}$$

- In case of a fair coin, probability of getting head or tail is same, **1/2 (0.5 or 50% chance)**, similarly, for a dice, chance of getting a certain number is **1/6**.

Odds

The odds of an event represent the ratio of the:

$$\frac{Probability\ that\ the\ event\ will\ occur}{Probability\ that\ the\ event\ will\ not\ occur} = \frac{P_{event}}{1 - P_{event}}$$

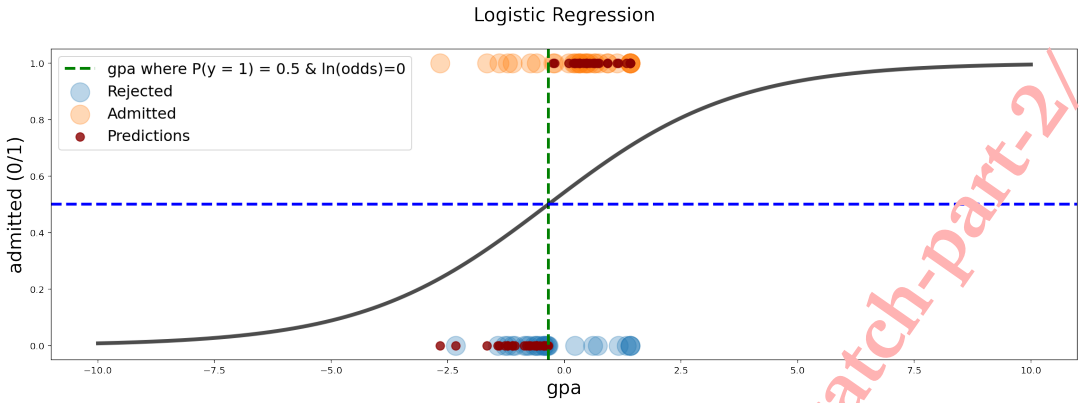
For example:

$$Odds\ of\ 3\ from\ a\ dice = \frac{Probability\ of\ getting\ 3}{1\ -\ Probability\ of\ getting\ 3} = \frac{1/6}{1 - 1/6} = \frac{1}{5}$$

So we can write:

```
fig
```

[17]:



[18]:

```
#X.gpa.mean()
```

[19]:

```
print("The logistic regression beta's are:")
print("beta_1 = {} and beta_0 = {}.".format(logR.coef_[0][0], logR.intercept_[0]))
```

```
The logistic regression beta's are:
beta_1 = 0.5005772297415476 and beta_0 = 0.17118[illegible]3.
```

Go to: L5: Logistic Regression

(L5-4-4) Interpretation of the Coefficients

Our feature (gpa) in X is standardized (0 mean, 1 variance), so the `gpa=0` indicates an average `gpa` and `gpa=1` indicates a value being one standard deviation larger than the mean, which is 0.

Meaning of the betas in log odds

The coefficients have a linear impact on the log-odds (recall the formula).

- If β_1 is 0, then β_0 represents the log odds of admittance for a student with an average gpa.
- β_1 is the effect of a unit increase in rescaled gpa on the log odds of admittance.

- **Log odds are hard to interpret**. Luckily though, we can apply the logistic transform to get the probability of admittance at different β values.
- From the curve in the above plot, we can see that values of gpa within 2 to 3 standard deviations of the mean lead to a practically linear increase of the probability of admission.
- The values very far to the left or the right hardly increase or decrease the probability of admission (s-shaped curve) any further as the curve becomes very flat.

Logistic regression coefficients can be exponentiated to get the odds ratio, and this is even easier to interpret the these coefficients. We will try this in the next lecture while working with titanic data set. In the mean time, these links could be useful to explore:

- Interpret coefficients – odd ratios in logistic regression
- exponentiate the logistic regression coefficients

Go to: L5: Logistic Regression

The β-coefficients are chosen in such a way that this **function is maximized**.

The optimal case would be that - the predicted probabilities for all class one observations are actually one - the predicted probabilities for all class zero observations are actually zero

There is not a closed-form solution to the beta coefficients like in linear regression, and the betas are found through optimization procedures.

If you are particularly interested in the math, these two resources are good:

A good blog post on the logistic regression beta coefficient derivation.

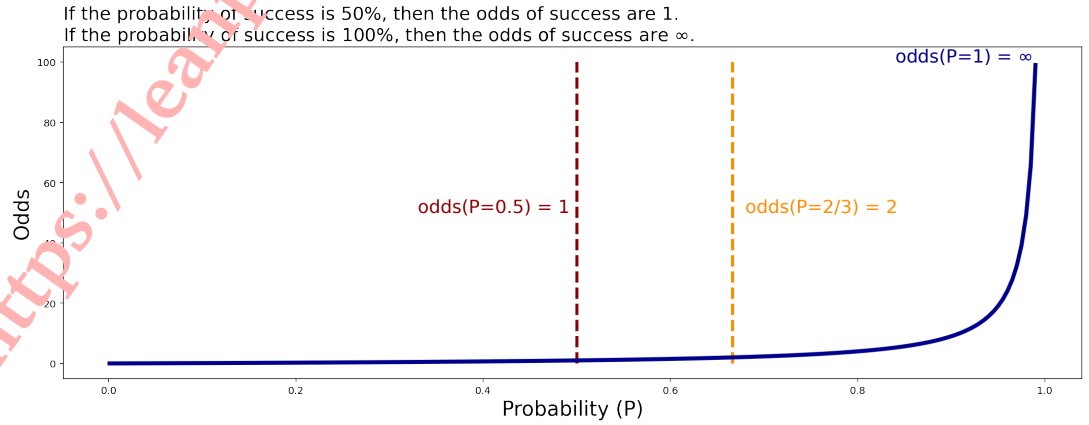
This paper is also a good reference.

Go to: L5: Logistic Regression

(L5-7-4) Illustration of a few functions

(L5-7-4-1) Probability vs Odds

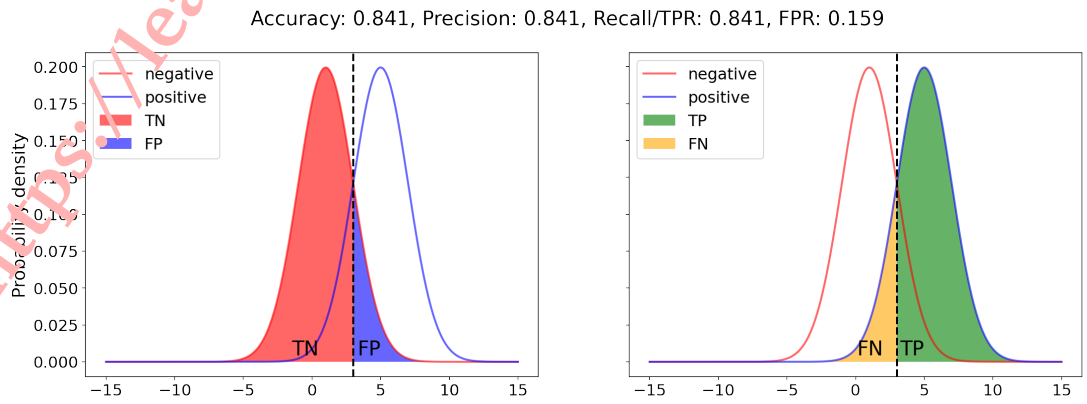
[34]:
```
# Function to calculate the odds of success.
def odds(p):return(p/(1-p))
# Generating a range of probabilities.
probabilities=np.linspace(0.001,0.99,200)
# Generate list of odds.
odds_list = [odds(proba) for proba in probabilities]
# Create figure.
plt.figure(figsize=(18,6))
# Plot blue line for odds as probability goes from 0.1% (0.001) to 99% (0.99).
plt.plot(probabilities,odds_list,lw=4,color='DarkBlue')
# Plot red dashed line to visualize odds when probability is 50%.
plt.vlines(0.5,0.0,100,ls="dashed",lw=3,color='DarkRed')
plt.text(0.33,50.0,"odds(P=0.5) = 1",fontsize=18,color='DarkRed')
# Plot orange dotted line to visualize odds when probability is 66.67%.
plt.vlines(0.6667,0.0,100,ls="--",lw=3,color='DarkOrange')
plt.text(0.68,50,"odds(P=2/3) = 2",fontsize=18,color='DarkOrange')
# Annotate blue line when probability is 100%.
plt.text(0.84,100,"odds(P=1) = $\infty$",fontsize=18,color='DarkBlue')

# Title, labels.....1
plt.title("If the probability of success is 50%, then the odds of success are 1.\n\
If the probability of success is 100%, then the odds of success are $\infty$.",
          ha='left',position=(0,1), fontsize=18)
plt.xlabel("Probability (P)",fontsize=20)
plt.ylabel("Odds",fontsize=20);
```



```
    ax[0].fill_between(x,y_1,0,where=(y_2<=odds*y_1),facecolor='r',alpha=0.
↪6,interpolate=True,label='TN')
    ax[0].fill_between(x,y_1,0,where=(y_2>=odds*y_1),facecolor='b',alpha=0.
↪6,interpolate=True,label='FP')
    ax[0].annotate('TN',xy=(5,0.005),xycoords='data',xytext=(threshold-2.5,0.015),
                   va="top",ha="right",fontsize=20)
    ax[0].annotate('FP',xy=(5,0.005),xycoords='data',xytext=(threshold+2,0.015),
                   va="top",ha="right",fontsize=20)
    ax[0].legend(fontsize=16);ax[0].tick_params(labelsize=16)
    ax[0].set_ylabel('Probability density', fontsize=18)

    ax[1].plot(x, y_1,'r-',lw=2,alpha=0.6,label='negative')
    ax[1].plot(x, y_2,'b-',lw=2,alpha=0.6,label='positive')
    ax[1].axvline(threshold,ls='--',lw=2,c='k')
    ax[1].fill_between(x,y_2,0,where=(y_2>=odds*y_1),facecolor='g',
                       alpha=0.6,interpolate=True,label='TP')
    ax[1].fill_between(x,y_2,0,where=(y_2<=odds*y_1),facecolor='orange',
                       alpha=0.6,interpolate=True,label='FN')
    ax[1].annotate('FN',xy=(5,0.005),xytext=(threshold-1.0
↪015),va="top",ha="right",fontsize=20)
    ax[1].annotate('TP',xy=(5,0.005),xytext=(threshold+2,0.
↪015),va="top",ha="right",fontsize=20)
    ax[1].legend(fontsize=16);ax[1].tick_params(labelsize=16)
    plt.show();return fig, ax
```

[38]:
```
# Initializing parameters
mean_1=1; mean_2=5; std_1=2; std_2=2; odds=
#Function call
fig, ax =␣
 ↪plot_confusion_matrix(loc_1=mean_1,loc_2=mean_2,scale_1=std_1,scale_2=std_2,odds=odds);
 ↪
#odds=1 then pr=0.5 == See the table above
```

```
Threshold: 2.998799879987999
TN: 0.841199505550696 FP: 0.158800494449304
FN: 0.1585101005404902 TP: 0.8414898994595098
Confusion matrix:
 [[0.841 0.159]
 [0.159 0.841]]
Accuracy:       0.841
Precision:      0.841
Recall/TPR:     0.841
FPR:            0.159
```



Logistic Regression

(End-to-end machine learning project)

– Predict if the person was dead or alive –

Author: Dr. Junaid Qazi, PhD

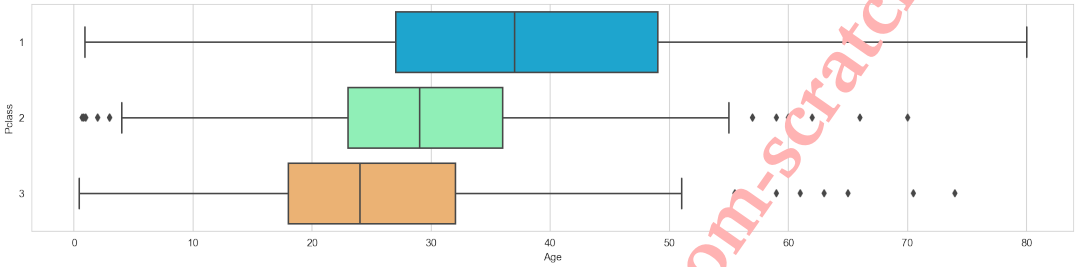
So, we know form EDA that some data is missing in our dataset, let's deal with that first.

`Age` **column is missing ~ 19.9% of its data.**

- A convenient way to fix `'Age'` column is by filling the missing data with `mean` or `average` value of all passengers in that column. **We can do even better** in this case, because we know that their are three passenger classes, **its better to use the average age for each missing passenger for its own class.**

Let's use a `boxplot()` to visually explore if their is any relationship in class and passenger age?

[17]:
```
plt.figure(figsize=(18, 4)) # setting the figure size, its subjective
sns.boxplot(x='Age',y='Pclass',data=train,palette='rainbow',orient='h');
```



Yes, `Pclass` and `Age` are somehow related, this makes sense, ***older the passenger is, higher the class he traveled in!***

So our hypothesis to to fill the missing `Age` with respect to the passenger class is the better way to fill in missing data in `Age` column!

We can write a function and use `apply()` from pandas for this task, however, before writing a function, we may want to know the average age of passengers for each class, `groupby()` **could be usefull here!**

Let's find average age of passengers in each class first, we only need `Pclass` and `Age` columns for this purpose!

[18]:
```
train[['Pclass','Age']].groupby('Pclass').mean() #describe() # try describe with␣
↪groupby!
```

[18]:
```
              Age
Pclass
1       38.233441
2       29.877630
3       25.140620
```

[19]:
```
train[['Pclass','Age']].groupby('Pclass').mean().loc[1]#.group_keys()
```

[19]:
```
Age    38.233441
Name: 1, dtype: float64
```

Now, we have average age for each class, let's write a custom function to fill the missing values in `Age` column. Super easy, we can use `if-else conditional statement` in the function!

[20]:
```
#Defining a function 'impute_age'
def impute_age(age_pclass): # passing age_pclass as ['Age', 'Pclass']

    # Passing age_pclass[0] which is 'Age' to variable 'Age'
    Age = age_pclass[0]
```

```
    # Passing age_pclass[1] which is 'Pclass' to variable 'Pclass'
    Pclass = age_pclass[1]

    #applying condition based on the Age and filling the missing data respectively
    if pd.isnull(Age):
        if Pclass == 1:return 38
        elif Pclass == 2:return 30
        else:return 25
    else:return Age
```

Let's apply the above function to our data now. We can use `apply()` method and pass `axis = 1` for column. (recall from pandas section)

[21]:
```
# grab age and apply the impute_age, our custom function
train['Age'] = train[['Age','Pclass']].apply(impute_age,axis=1)
# You may want to revise 'impute_age' function and the statement above!
```

Let's try to re-plot the heatmap now, after fixing the age column!

[22]:
```
plt.figure(figsize = (18,6)) # just fig size
sns.heatmap(train.isnull(),yticklabels=False,cbar=False,cmap='viridis');
```

So, we got this done, ***no more yellow color in the Age*** column. This means we have filled all the missing values in `Age` column using `impute_age` function.

Now, there is another column `Cabin` **with ~ 77.1% of missing data.**

77% is lots of information! Well, we might be able to analyze the ticket number to see if we can get some information on the Cabin, however, let's leave it at the moment and simply drop this column.

[23]:
```
# dropping 'Cabin' column, axis =1 for column and inplace = True for permanent␣
↪change!
train.drop('Cabin',axis=1,inplace=True)
```

Let's see how the `heatmap` looks like now!

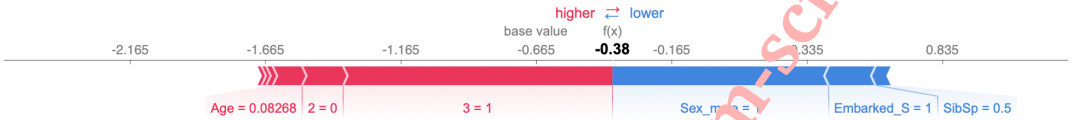
[24]:
```
plt.figure(figsize = (18,6))
sns.heatmap(train.isnull(), yticklabels=False, cbar=False, cmap='viridis');
```

[74]:
```
# The base value, reference value that the feature contributions start from.
print("The base value is: ", explainer_shap.expected_value)
```

```
The base value is:  -0.6649791332453798
```

[75]:
```
# Explaining the ith (see LIME part for the value of i=10) observation.
# Visualize the ith prediction's explanation (use matplotlib=True to avoid
 ↪Javascript)
shap.force_plot(base_value=explainer_shap.expected_value,
                shap_values=shap_values[i,:],
                features=X_train_s.iloc[i,:])#,
                #matplotlib=True)
```

[75]:
```
<shap.plots._force.AdditiveForceVisualizer at 0x7fc0618eb370>
```



So, the values in red are helping the model to make prediction of class 0 (died) in this observation, whereas the blue are reducing the chances to predict class 0 and helping to predict class 1.

This link could be useful to explore further on SHAP with Logistic Regression

Go to: L6: Logistic Regression – Titanic data

[76]:
```
# Try this interactive force plot
#shap.force_plot(base_value=explainer_shap.
 ↪expected_value,shap_values=shap_values,features=X_train)
```

(L6-7) To do

Considering the amount of data and time we have used in this project, the results are very good. The can be improved with more data and adding more features.

Few thing that you may want to consider while practicing:** * Well, we considered `Pclass` as a categorical column and created its dummies, try to re-train the model with the original `Pclass` column without dummies and compare your results. What are you findings and why the results are different?

- Do you think that you can get any information from the `Ticket` or any other column.
- Grab the prefix/title (Mr. Mrs. Dr. etc) from `Name` as a feature

Titanic dataset is very popular for Classification problem and their are number of good kernels on kaggle. Check the Python ones, you may get different ideas to improve your model.

The kernels in other languages such as **R** are also useful, you can get an idea on data cleaning, plotting and some feature engineering that you can implement in Python as well.

Go to: L6: Logistic Regression – Titanic data

Good Luck!

(L6-8) Recommended Readings

- Amazing explanation oN Logistic Regression – Why sigmoid function?
- Why is logistic regression considered a linear model?
- Feature Importance

Logistic Regression for multiclass classification

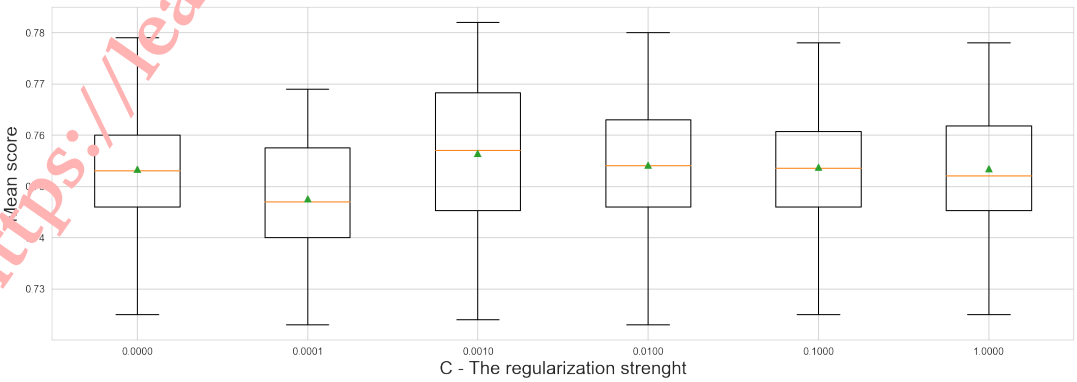
(End-to-end machine learning project - hands-on)

– Working with more than two classes –

Author: Dr. Junaid Qazi, PhD

```
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=5, random_state=101) #␣
 ↪creating instance
    # Stratification is the process of dividing members of the population into
    # homogeneous subgroups before sampling.
    # evaluate the model
    scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
    return scores

# define dataset
X, y = the_dataset()
# get the models to evaluate
models = the_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    # evaluate the model and collect the scores
    scores = model_eval(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize progress along the way
    print("C = {}; mean_score = {}; std = {} ".format(
        name,round(np.mean(scores),4),round(np.std(scores),4)))
print("\nTotal compute time (sec) = {}".format(time.time() - start_time))
# plot model performance for comparison
plt.figure(figsize=(18,6))
plt.xlabel("C - The regularization strenght", fontsize=18);plt.ylabel("Mean score",␣
 ↪fontsize=18)
plt.boxplot(results, labels=names, showmeans=True);
```

```
Data (X, y) is created...
list of C (inverse of regularization strength) values:  [0.0, 0.0001, 0.001,
0.01, 0.1, 1.0]
Training multinomial logistic regression models...!

C = 0.0000; mean_score = 0.7533; std = 0.0133
C = 0.0001; mean_score = 0.7476; std = 0.0125
C = 0.0010; mean_score = 0.7564; std = 0.0142
C = 0.0100; mean_score = 0.7541; std = 0.0135
C = 0.1000; mean_score = 0.7537; std = 0.0131
C = 1.0000; mean_score = 0.7534; std = 0.0132

Total compute time (sec) = 44.79873824119568
```



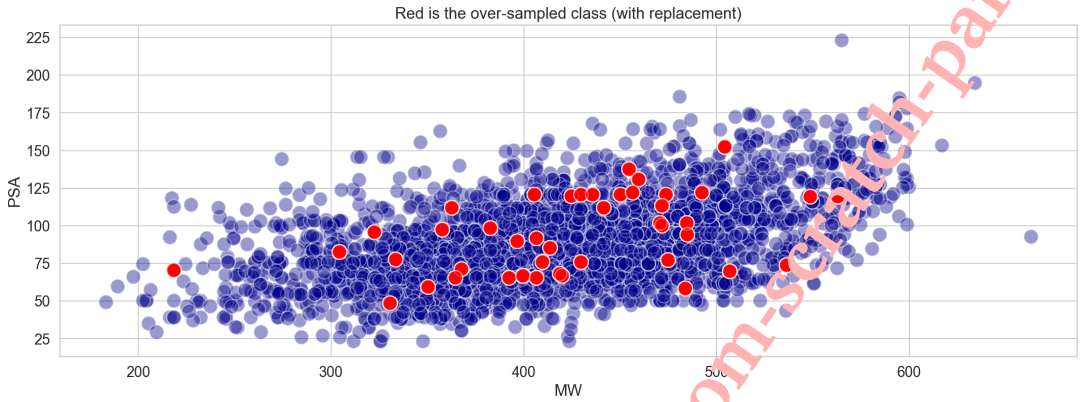
Working with imbalance class in the data

(Theory and hands-on)

– Bioassay data with class imbalance –

Author: Dr. Junaid Qazi, PhD

[24]:
```
# Let's visualize the imbalance! -- Any two features to get a scatter plot
plt.figure(figsize=(18,6))
sns.scatterplot(data=over_sampled[over_sampled.Outcome=='Inactive'],
                x='MW',y='PSA',s=200,alpha=0.4,color='DarkBlue')
sns.scatterplot(data=over_sampled[over_sampled.Outcome=='Active'],
                x='MW',y='PSA',s=200, alpha=0.4,color='red');
plt.title("Red is the over-sampled class (with replacement)");
```



Well, its creating a copy of same observations for minority class, we will not see anything in a scatter plot as they they are overlapped!

[25]:
```
X_os=over_sampled.drop('Outcome',axis=1)
y_os=over_sampled.Outcome
#y_os=test.drop('Outcome',axis=1)#;y_test=test.Outcome
```

[26]:
```
# Try yourself and compare
#scaler = StandardScaler()
#X_os = scaler.fit_transform(X_os)
```

[27]:
```
X_os_train, X_os_test, y_os_train, y_os_test =␣
 ↪train_test_split(X_os,y_os,test_size=0.3,random_state=101)
```

The training set (X_os_train, y_os_train)

[28]:
```
# Creating model instances
logR_os = LogisticRegression(max_iter=10000)#multi_class='ovr') # one-vs-rest
# fitting the model
logR_os.fit(X_os_train,y_os_train)
# Accuracy Score
print("Score when multi_class='ovr':",logR_os.score(X_train,y_train))
```

```
Score when multi_class='ovr': 0.8706176961602671
```

[29]:
```
# predictions
print("Data is: (X_os_train, y_os_train)")
print("Over-sampled the minority class in this dataset.\n")
pred_os_train = logR_os.predict(X_os_train)
# Confusion matrix and classification report
print("The confusion matrix:")
print(metrics.confusion_matrix(y_os_train,pred_os_train))#,labels=[0, 1, 2]))
print("\nThe classification report:")
print(metrics.classification_report(y_os_train,pred_os_train))#,labels=[0, 1, 2]))
```

```
0.7.0
```

We already have features in X and the target in y from the original dataframe. Let's import `SMOTE()`, create its instance and generate synthetic data!

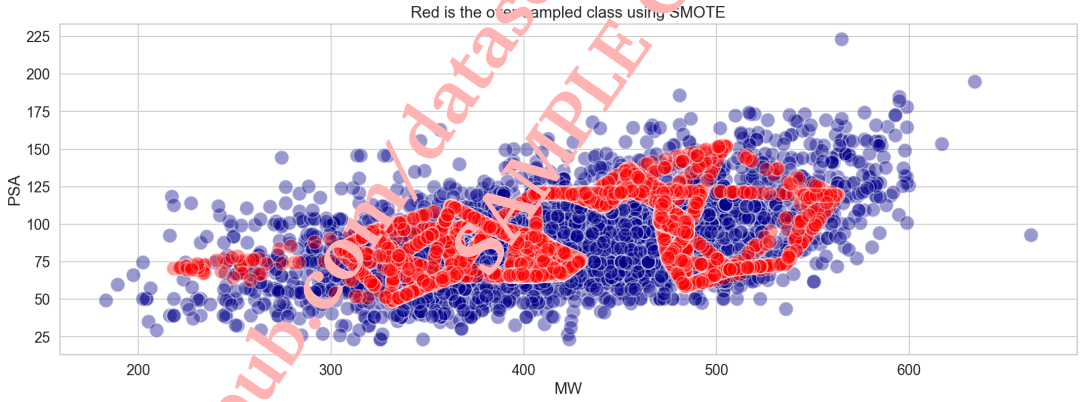
[33]:
```
from imblearn.over_sampling import SMOTE
# Using default parameters in SMOTE
smote = SMOTE()# <shit+tab> and exploresampling_strategy=1)
X_smote, y_smote = smote.fit_resample(X, y)
```

[34]:
```
#imblearn.over_sampling.
```

[35]:
```
y_smote.value_counts()
```

[35]:
```
Active      3375
Inactive    3375
Name: Outcome, dtype: int64
```

[36]:
```
# Let's visualize the imbalance! -- Any two features to get a scatter plot
# Creating an intermediate dataframe for this visualizations only
df_smote=X_smote.copy()
df_smote['Outcome']=y_smote
plt.figure(figsize=(18,6))
sns.scatterplot(data=df_smote[df_smote.
 ↪Outcome=='Inactive'],x='MW',y='PSA',s=200,alpha=0.4,color='DarkBlue')
sns.scatterplot(data=df_smote[df_smote.Outcome=='Active'],x='MW',y='PSA',s=200,␣
 ↪alpha=0.4,color='Red')
plt.title("Red is the over-sampled class using SMOTE")
```



We can see, the synthetic samples are not just the copies of the same observations now!

Go to: L8: Handling imbalanced classes in the dataset

[37]:
```
# Try yourself and compare
#scaler = StandardScaler()
#X_smote = scaler.fit_transform(X_smote)
```

[38]:
```
X_smote_train, X_smote_test, y_smote_train, y_smote_test = train_test_split(
    X_smote,y_smote,test_size=0.3,random_state=101)
```

[39]:
```
#X_train=train.drop("Outcome", axis=1)
#y_train=train.Outcome
```

  - Our sample could be different for different variables and would be difficult to compare the analysis because of sample differences every time.

- Single imputation methods
  - mean, median or mode substitution – a kind of complete case analysis. It ignores the relationships between variables and weakens covariance and correlation estimates. Such assumptions can reduce variability in the data.
  - dummy variable control (1 is missing 0 if available) – One of the advantage is we use all the available information, however, this results in biased estimates (in case, there is a legitimate skip, these is no bias)
  - simple regression – Well, we use the information from available data and the model fit to estimate the missing data. This could overestimates the model fit and correlation estimates along with weakening the variance.
- Model-based techniques – Advanced
  - Maximum likelihood – the value that most likely to be observed by identifying a set of parameters that produces the highest log-likelihood
  - Multiple imputation using specified regression models – n repetitions results in separate dataset every time – – more accurate variability – cumbersome coding
  - Model based clustering

Want to know more, search for model-based imputation methods and you will find nice publications.

Missing data: Our view of the state of the art – 2002 is one of the great read by Joseph L. Schafer and John W. Graham. 2nd link for pdf copy

Scikit-learn provides useful modules to impute the missing values as well!

Also good to search for these terms to understand some terminology for the missing data – MCAR (Missing Completely at Random), MAR (Missing at Random), and NMAR (Missing not at Random)……! You can also find explanation in any data mining book!

I think, this is enough on missing data and ways to handle it.

Go to: L9: Predicting Chronic Kidney Disease

(L9-3-4) Complete case analysis Let's look at our ckd data and see how the data look like if we try using **listwise deletion – A complete case analysis**!

[8]:
```
# Dropping any observation that has a missing value
print("Available data with number of complete cases -- listwise deletion")
print("Total number of observations in the original data: {}".format(len(ckd)))
print("We will left with {} observations and {} columns.".format(
    ckd.dropna().shape[0],ckd.dropna().shape[1]))
print("% of data loss: {}".format(100-ckd.dropna().shape[0]/len(ckd)*100))
print("\nClass distribution for complete case analysis:")
print(ckd.dropna()['class'].value_counts())
```

```
Available data with number of complete cases -- listwise deletion
Total number of observations in the original data: 400
We will left with 158 observations and 25 columns.
% of data loss: 60.5

Class distribution for complete case analysis:
notckd    115
ckd        43
Name: class, dtype: int64
```

Well, the above numbers are not very encouraging. We can't really trust on our model trained on complete cases only as we are losing more than 60% of the data in this situation. The class balance got bad as well, now we have ~27.2% of the ckd class, this was (250/400)*100 = 62.5% in the original data. Actually, most of the data loss is from ckd class (100*(250-43)/250 = 82.8%).

K-nearest neighbors (KNN)

(Working principle – hands-on)

– We will talk about theory and learn by doing

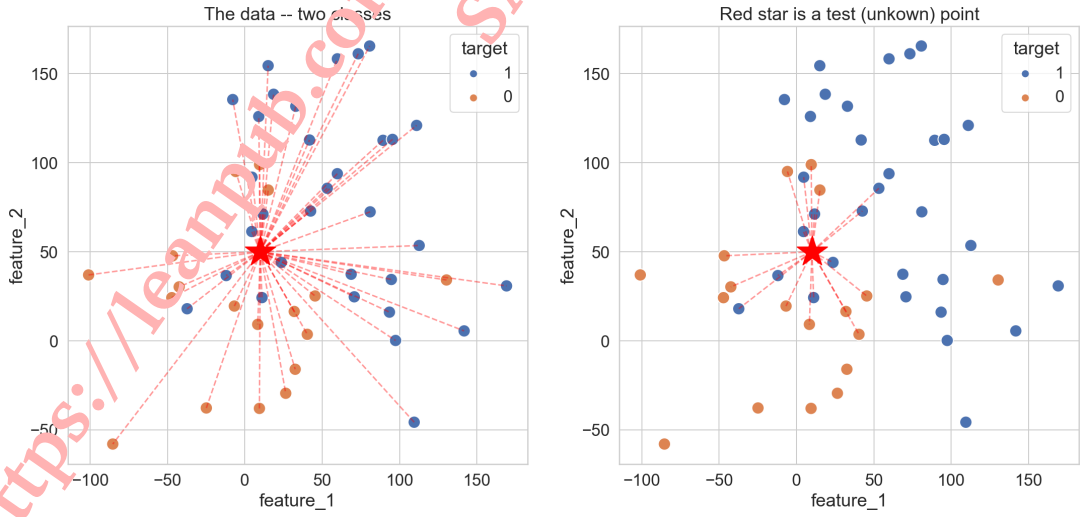
Author: Dr. Junaid Qazi, PhD

```
    try:
        print("{} votes from target class {}".format(knn_votes.values[0], knn_votes.
↪index[0]))
    except:
        print("All votes belongs to the other class")
    try:
        print("{} votes from target class {}".format(knn_votes.values[1], knn_votes.
↪index[1]))
    except:
        print("All votes belongs to the other class")
    print("The test data belongs to the majority class")
    # we can simply use head with k to grab first 13 rows!
    for index,feature_2,feature_1,target,distance in df.sort_values("distance").
↪head(k).to_records():
        ax2.
↪plot([test_point[0][0],feature_1],[test_point[0][1],feature_2],'--',alpha=0.
↪4,color='red')
    return None
```

Go to: L10: K-nearest neighbors (KNN) – Working principle hands-on

[7]:
```
# Function call with selected k value
plot_knn_distances(k=20)
# To Do: Try different number for k and see what class you are getting!
# Try modifying the function to store the output class for test point with␣
↪different k
```

```
All the distances from unknow point are calculated and plotted on the left plot
in dotted lines.
The selected value (user input) of k is 20
Found ties...!
Ties resolves by selecting k to odd number 19
10 votes from target class 0
9 votes from target class 1
The test data belongs to the majority class
```



Now, we have a complete understanding of knn algorithm and its working principle. It is one of the simplest algorithm for classification.

From the summary statistics, using `describe()`, we can see that the features such as, `Cd_1`, `Cd_5`, `Cd_6` and `Cd_9`, are on very different scales. The data is coded, we even don't know what are they representing!

Let's grab the mean and standard deviations (std) of all features and plot them to see how they look like. What we can do, we can call `Transpose` on `describe()` and grab the required measures.

[6]:
```
means_stds = df.describe().T[['mean', 'std']]
means_stds
```

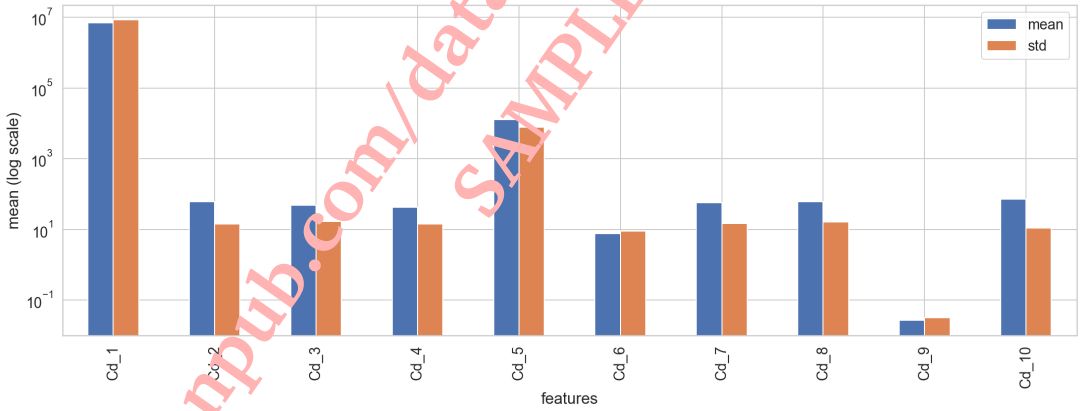
[6]:

	mean	std
Cd_1	7.096316e+06	8.398082e+06
Cd_2	6.077210e+01	1.401729e+01
Cd_3	4.842460e+01	1.692452e+01
Cd_4	4.172640e+01	1.404612e+01
Cd_5	1.271827e+04	7.889550e+03
Cd_6	7.485882e+00	8.803856e+00
Cd_7	5.623460e+01	1.489099e+01
Cd_8	6.004460e+01	1.618612e+01
Cd_9	2.633790e-02	3.116939e-02
Cd_10	7.195530e+01	1.078270e+01

It's always a good idea to visualize the data, we can get a bar plot for mean values of the features in our dataset.

I am going to use log-scale along y for means.

[7]:
```
means_stds.plot(kind='bar',figsize=(18,6));
plt.yscale('log') # try linear scale by removing this line!
plt.xlabel('features')
plt.ylabel('mean (log scale)');
```



Notice the log scale along y-axis, we can clearly see significant variations in the scales. `Notice, Cd_1 vs Cd_9!`

Go to: L14: K Nearest Neighbors - hands-on implementation

Let's see how the data look like in a pairplot!

We can do more visualization using range of plotting options to understand the data. However, in this lecture, machine learning is the focus, so we can give a quick overview using pariplot and move on to the KNN algorithm.

[8]:
```
sns.pairplot(df, hue='Result');
```

****

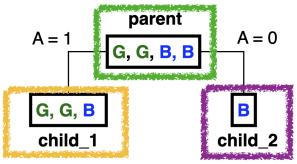
****

Decision Trees - Splitting at nodes

Entropy & Information Gain

A	B	C	Target
1	1	1	G
1	1	0	G
0	0	1	B
1	0	0	B

Splitting at A:



H (parent) $= -(2/4)\log_2(2/4) - (2/4)\log_2(2/4)$
$= -(0.5)(-1) - (0.5)(-1) = 1$

H (child_1) $= -(1/3)\log_2(1/3) - (2/3)\log_2(2/3)$
$= -(1/3)(-1.58) - (2/3)(-0.58) = 0.92$

H (child_2) $= -(1)\log_2(1) = 0$

IG = H (parent) – (3/4) H (child_1) – (1/4) H (child_2)
IG = 1 – (3/4) 0.92 – (1/4) 0 = 1 – 0.69 – 0 = **0.31**

****

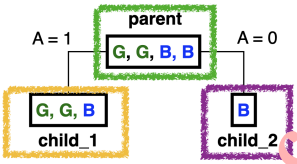
****

Decision Trees - Splitting at nodes

Entropy & Information Gain

A	B	C	Target
1	1	1	G
1	1	0	G
0	0	1	B
1	0	0	B

Splitting at A:



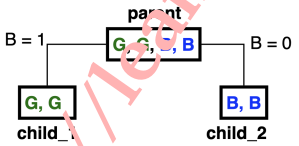
H (parent) $= -(2/4)\log_2(2/4) - (2/4)\log_2(2/4)$
$= -(0.5)(-1) - (0.5)(-1) = 1$

H (child_1) $= -(1/3)\log_2(1/3) - (2/3)\log_2(2/3)$
$= -(1/3)(-1.58) - (2/3)(-0.58) = 0.92$

H (child_2) $= -(1)\log_2(1) = 0$

IG = H (parent) – (3/4) H (child_1) – (1/4) H (child_2)
IG = 1 – (3/4) 0.92 – (1/4) 0 = 1 – 0.69 – 0 = **0.31**

Splitting at B:



H (parent) $= -(2/4)\log_2(2/4) - (2/4)\log_2(2/4) = 1$
H (child_1) $= -(2/2)\log_2(2/2) = 0$
H (child_2) $= -(2/2)\log_2(2/2) = 0$
IG = H (parent) – (2/4) H (child_1) – (2/4) H (child_2)
IG = 1 – 0 – 0 = **1**

****

****

(L13-3-4) (OPTIONAL) Extremely Randomized Trees (ExtraTrees)

Moving forward, it might be a good idea to know little bit about Extremely randomized trees as well. This is another tree-based ensemble method for supervised classification and regression problems, which was introduced by Pierre Geurts, Damien Ernst and Louis Wehenkel in their article in 2006 "Extremely randomized trees". The algorithm of growing Extremely randomized trees is similar to Random Trees/Forest, but there are two differences:

- Extremely randomized trees don't apply the bagging procedure to construct a set of training samples for each tree. The same input training set is used to train all trees. ==> *Note - we can still add bagging layer if we want!*
- Extremely randomized trees pick a node split very extremely (both a variable index and variable splitting value are chosen randomly), whereas Random Forest finds the best split (optimal one by variable index and variable splitting value) among random subset of variables.

==> Adding one more step of randomization (and thus de-correlation) yields extremely randomized trees, or *ExtraTrees*. These can be trained using bagging (sampling of observations) and the random subspace method (sampling of features), along with an additional layer of randomness. **Instead of computing the locally optimal feature/split combination (based on, e.g., information gain or the Gini impurity) for each feature under consideration, a random value is selected for the split. This value is selected from the feature's empirical range.** ==>*This further reduces the variance, but causes an increase in bias. If you're considering using ExtraTrees, you might consider this to be a hyperparameter you can tune.*

- `RandomForestClassifier` and `ExtraTreesClassifier` on sci-kitlearn
- `DecisionTreeClassifier vs ExtraTreeClassifier` useful link on stackoverflow

scikit-learn provides a convenient way to implement extremely randomized trees using its module `ExtraTreesClassifier`. This class implements a meta-estimator (*an estimator which takes another estimator as a parameter, e.g. GridSearchCV)* that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. - *`bootstrap=False` by default*

Go to: L13: Decision trees and tree based ensemble learning

[32]:
```
from sklearn.ensemble import ExtraTreesClassifier
```

[33]:
```
# Creatinbg model instance with 100 trees
et = ExtraTreesClassifier(n_estimators=100) # bootstrap=False by default␣
 ↪#max_features='auto',
```

Just a quick note: In ExtraTrees, intrinsically, the randomness does not come from bootstrapping the data, but rather comes from the random splits of all observations. However, we can use bootstrap to add extra randomness.

[34]:
```
et = et.fit(X_train, y_train) # training the model
```

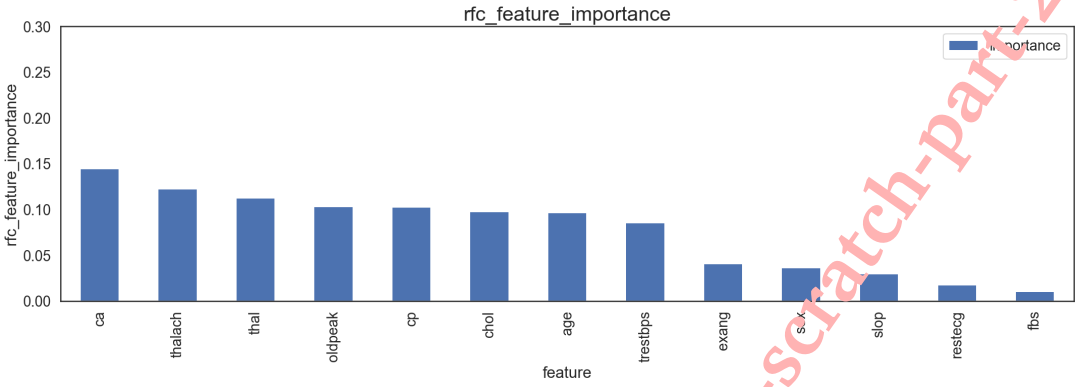
[35]:
```
pred_et = et.predict(X_test) # getting predictions
```

[36]:
```
print('Confusion matrix for Extra Trees on test data:')
print(confusion_matrix(y_test, pred_et))

print('\nClassification report for Extra Trees on test data:')
print(classification_report(y_test, pred_et))
```

```
Confusion matrix for Extra Trees on test data:
[[41  5]
 [ 7 34]]

Classification report for Extra Trees on test data:
              precision    recall  f1-score   support
```

of samples it splits. Another commonly used approach for the tree split is calculating the Information Gain which depends upon entropy.

[43]:
```
# single line code using our custom function
feature_plot(rfc.feature_importances_, features_names, 'rfc_feature_importance')
```

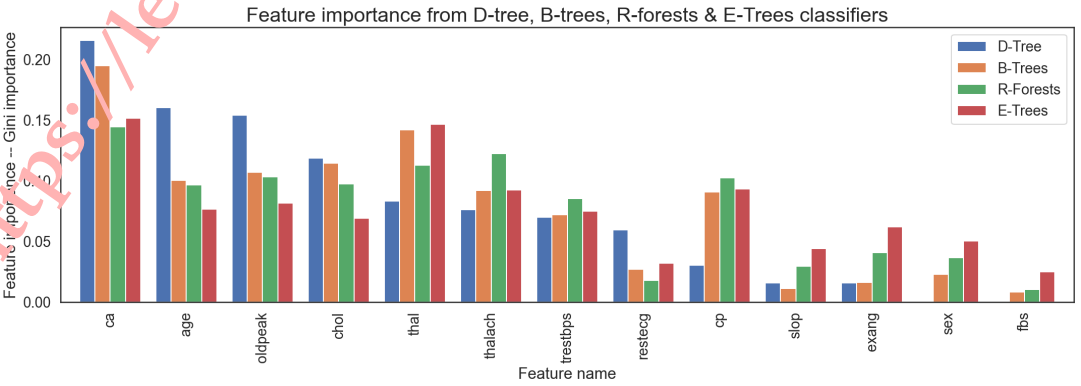


Go to: L13: Decision trees and tree based ensemble learning

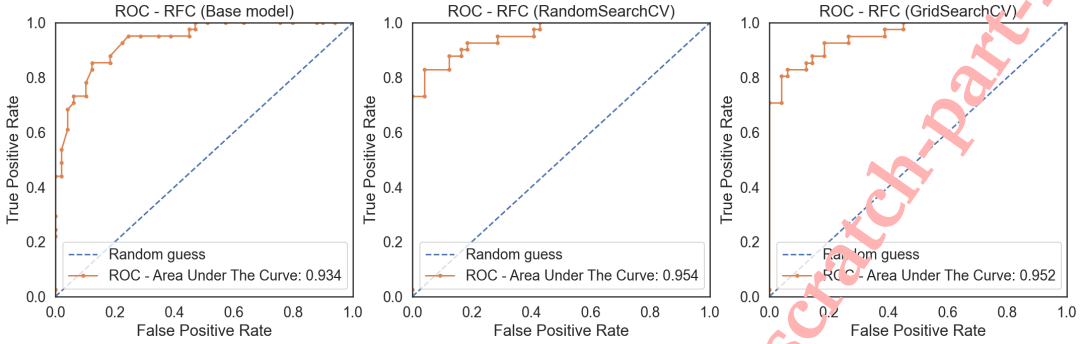
(L13-4-4) Comparing feature importance

(OPTIONAL) – Let's create a dataframe with feature importance from our trained models, `D-Tree`, `B-Tree`, `R-Forests` and `E-Trees`. We can get a bar plot for comparisons and discuss.

[44]:
```
# Creating dataframe with feature importance from rfc and dtree
feature_imp = pd.DataFrame(dtree.feature_importances_.T, columns =␣
 ↪['D-Tree'],index=features_names)
feature_imp["B-Trees"]=bagged_trees.feature_importances
feature_imp["R-Forests"]=rfc.feature_importances_
feature_imp["E-Trees"]=et.feature_importances_
feature_imp.sort_values(by = ['D-Tree'], ascending = False, inplace = True) # good␣
 ↪to sort!
#feature_imp.head()
# getting bar plot for feature importance
feature_imp.plot(kind = 'bar',width=0.8, figsize = (18,5))
plt.title("Feature importance from D-tree, B-trees, R-forests & E-Trees␣
 ↪classifiers", fontsize=20)
plt.xlabel("Feature name")
plt.ylabel("Feature importance -- Gini importance");
# gini importance is not same as gini index!
```



```
plot_rocs(y_test_01, rfc_prob, ROC_area_rfc, ax[0], 'ROC - RFC (Base model)')
plot_rocs(y_test_01, random_search_rfc_prob, ROC_area_random_search_rfc, ax[1],␣
 ↪'ROC - RFC (RandomSearchCV)')
plot_rocs(y_test_01, grid_search_rfc_prob, ROC_area_grid_search_rfc, ax[2], 'ROC -␣
 ↪RFC (GridSearchCV)')
```



- What do you learn from the above ROC-Curves?
- Which model you will finally select?

Do you think you can find even better model? Try different `n_estimators` along with different set of parameters in your hyper-parameter tuning ..... what else you can do?

Go to: L13: Decision trees and tree based ensemble learning

(L13-7) Playing with probability cut-off

Let's consider, we are happy with the model after tuning with grid-search module. Now, the question is, should we stay with the probability cut-off 0.5 for the class prediction?

Sometimes, we are interested in different probability cut-off for the class predictions, typically in the medical related studies. We want to bring the high-risk patients for screening at very early stage to avoid complications and reduce the long term cost for treatments.

Let's look at the prediction from our selected model `grid_search_rfc_pred` once again!

[77]:
```
print(y_test.value_counts())
```

```
N    49
Y    41
Name: target, dtype: int64
```

[78]:
```
tn, fp, fn, tp = confusion_matrix(y_test,grid_search_rfc_pred).ravel()
print("tn: {} fp: {} ".format(tn,fp))
print("fn: {} tp: {}".format(fn,tp))
```

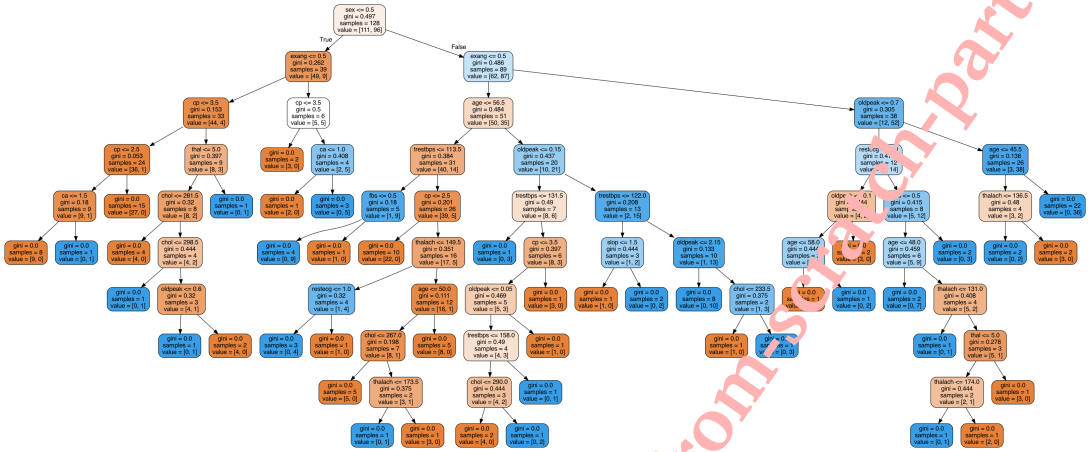
```
tn: 46 fp: 3
fn: 7 tp: 34
```

So, if we look at the above confusion matrix, the model is returning 7 false negative and 3 false positive. So, 7 patients actually have a heart problem and the model is not able to predict. This is actually putting these patients in real problem for future.

What if we change the probability cut-off.....say, `0.3` and see if we can reduce the false negative cases, right? False positive may increase, however it is not dangerous as they will be screened for any potential risk.

```
graph = pydot.graph_from_dot_data(dot_data.getvalue())
Image(graph[0].create_png())

# We bootstrapped the features, our datapoints will stay same.
# For example in the root_node, we have in total 207 data points (same as in our␣
↪training data)
```

[88]:



- In the same way, we can grab trees from other models that we have trained above, try yourself!
- We can always use `SHAP` and `LIME` libraries to explain the model, please consult the previous lectures and try yourself....!

This was all about the tree based ensemble learning at the moment. Use your own data and trained classifier for all the models that we have learned for classification so far. Compare them and see which one should be your final selection. Recall – `No Free Lunch Theorem`!

Go to: L13: Decision trees and tree based ensemble learning

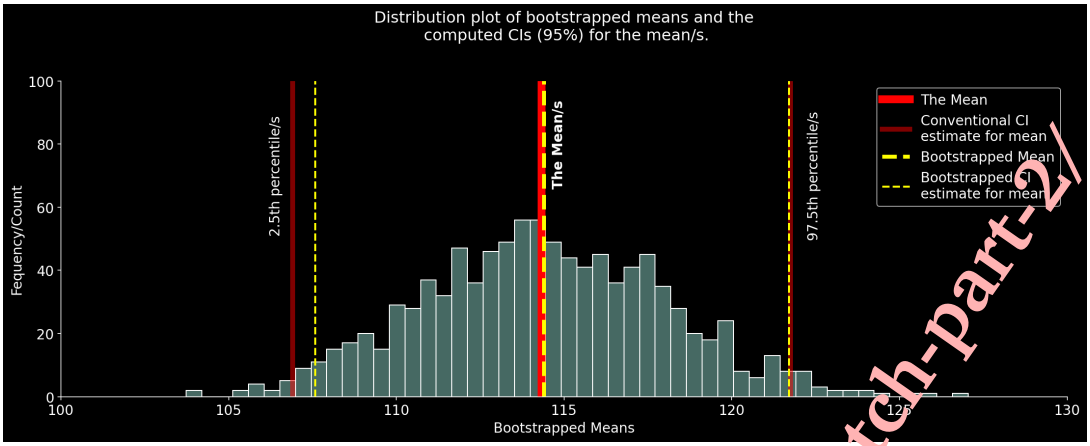
(L13-10) (OPTIONAL) coding practice for fun

Coding is fun and we must keep practicing to sharpen our skills. In your free time, think about some task, e.g. writing your own function for value_counts, nuique, head or groupby .....it is always helpful. Let's write a write a function for bootstrapping.

Try it before you look at the given below! **It would be a good practice for loops and to understand the bootstrapping.**

[89]:

```
def boot_strap(features, n_sub_samples=10, n_features=13, replacement=True):

    """
    This function takes following arguments:
        replacement=True -- default value
        features = features_dataframe
        n_sub_samples = 10 no. of required subsamples
        n_features = 13 no. of random features in each sub sample less or equal to␣
↪columns in features.

    The function returns a list of dataframes after bootstrapping.
    Please note, that the assumptions is uniform distribution over all
    featrues (no probabilities are provided in np.choice in the while loop)."""
    if (n_features > len(features.columns)):
        print("Requeted feature in each bootstrapped sample is more than number of␣
↪feature in the data")
```



Go to: L14: Bootstrapping and Confidence Interval

(L14-8) Confidence interval for the median – A more practical example

In the case of estimating the confidence around the sample mean, the bootstrapping procedure may not be particularly useful since our sample mean has nicer distributional properties (see above distribution plot of APM).

Mean and median are often presented both as descriptive statistics, however the median is a central value of the data. It is that value for which one expects half of the (possible or observed) values being smaller and the other half being larger.

The bootstrap becomes much more useful **when we need to calculate our uncertainty around statistics without straightforward formulas** or ones with unreasonably strict assumptions. **The median** is one such statistic.

Standard error (SE) of medians is ~ 25% greater than standard error of mean and the formula and can be written as:

$$\text{S.E. median} \approx \text{S.E. mean} + 0.25 \cdot \text{S.E. mean}$$

$$\text{S.E. median} = 1.2533 \cdot \text{S.E. mean}$$

The above equation is a function of the **Standard Error (S.E.) of the mean** and uses a heuristic multiplier `1.2533`. Furthermore, it requires these assumptions to work:

1. n is large (large number of observations)
2. The sample of measurements are drawn from a normally distributed population

What if these assumptions is impractical?

The second assumption is strict – many distributions are not normal. Well, the median is much more useful when we suspect a non-normally distributed population.

==> ***Do you know?*** *The mean, median, and mode of a normal distribution are equal. The area under the normal curve is equal to 1.0.* What do you think, what is the benefit to calculate the median over the mean if we know ahead of time that the population is normally distributed?

Go to: L14: Bootstrapping and Confidence Interval

Support Vector Machines

(Theory and hands-n)

– We will talk about the working principle and visualize the effects –

Author: Dr. Junaid Qazi, PhD

(L15-1) A recall on regression for classification

Recall the regression based algorithms, in **linear regression**, we use a best fitted line to predict a continuous target. >What if we try to use the linear regression algorithm to predict some class/es (e.g. 0/1), it does not sound like a good idea! *(we typically convert the categorical targets/labels to integer classes (0/1).)*

Well, to accomplish a classification task, we would rather think about the line as a boundary that splits the space instead of fitting the points.

Here it comes the **logistic regression**, where we have learned how transfer functions can be used to tackle the classification problem using linear regression based classification algorithm. The logistic transfer function converts the real numbers to the probabilities and the algorithm then make use of these class probabilities as a proxy for the class predictions. Well, this created a boundary, as we are tackling a problem where the answer is no/yes or 0/1 by solving the logistic regression problem, and our typical old loss functions (MSE, MAE etc) are not helpful. We had to use log-loss or binary cross-entropy loss function to get this done. *see eq. 5.10 in section 5.3 - The cross-entropy loss function*

> Cross-entropy loss function returns a score that summarizes the average difference between the actual and predicted probability distributions for predicting class 1. The score is minimized and a perfect cross-entropy value is 0 for the accuracy of 1.

We said **the best possible line as a boundary**, and ideally we expect this boundary to make as few classification mistakes as possible. For evaluation, we look for the true label/s of the class; if it matched to the predicted label, the loss is 0 and if it does not match, the loss is 1. If we think about this misclassification loss, we actually are treating every misclassification equally bad. However, some classifications could be worse then others (based on how close they are to the decision boundary). We can imagine, how bad such linear separation for the two classes could be. ***Can we think about some alternate approach to deal with "equally bad" as "how bad"?***

Let's see how SVMs is helpful!

Go to: L15: Introduction to SVMs

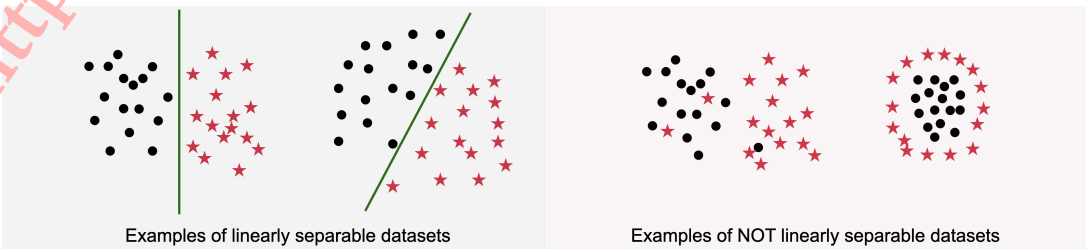
(L15-2) Support Vector Machine – The SVM

What if we could think about treating a misclassification/s based on `how bad` they have been misclassified, like impose stronger penalties for the observations that are found deeper in the territory of the other class. Farther the misclassification/s are from the decision boundary, the more wrong they are, thus deserves a higher penalty. In fact we would not mind a margin for error as well and even correctly classified observations that are really close to the boundary (almost misclassified) could contribute to the penalty. Make sense, right!

Well, this is where the Support Vector Machine (SVM) algorithm comes in, which follows a different approach for classification. The algorithm still fits a decision boundary like a logistic regression, but uses a different loss function called "The Hinge Loss", an alternative to cross-entropy for binary classification problems and used for maximum-margin classification. It is primarily developed to use with Support Vector Machine (SVM) models, and for binary class classification where the targets are `-1/1`. The function encourages the observations to have the correct sign while assigning more error where there is a sign difference between the true and predicted class labels.

(L15-3) How does the SVM classify?

In the figure below, we have two types of example datasets for classification; linearly separable and not linearly separable.



(L15-8) The hinge loss and non-linearly separable cases

Consider the case when there is no line/plane that can separate all the observations perfectly, here, we need to introduce the capacity for model error. With the constraint "$y \cdot (w^T \cdot X + b) \geq 1$" (also given above), we can introduce the **hinge loss function**:

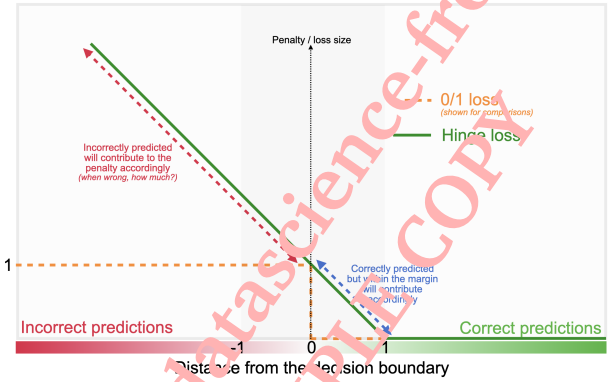
$$\text{hinge loss} = \sum_{i=1}^{n} \max\left(0, 1 - y_i \cdot (w^T \cdot x_i + b)\right)$$

$$\text{as } f(x_i) = (w^T \cdot x_i + b)$$

$$\text{hinge loss} = \sum_{i=1}^{n} max\left(0, 1 - y_i \cdot f(x_i)\right)$$

Where as, **for correct prediction/s**, when:

- $f(x_i) > 1$: the point lies *outside* the margin and **does not** contribute to the loss.
- $f(x_i) = 1$: the point is *on* the margin and **does not** contribute to the loss.
- $f(x_i) < 1$: the point lies *inside* the margin and **does** contribute to the loss accordingly.



So, in general, whenever $f(x_i)$ is NOT ≥ 1, the function (penalty) will be greater than zero and contribute to the loss accordingly for the respective datapoint/observation.

Go to: L15: Introduction to SVMs

(L15-9) Hinge loss and "slack"

Suppose a situation where it is not possible to perfectly separate the classes, the hinge loss with a regularization parameter C is helpful:

$$\min ||w||^2 + C \sum_{i=1}^{N} \epsilon_i \quad \text{subject to} \quad y_i(w^T x_i + b) \geq 1 - \epsilon_i$$

ϵ_i is the errors from the algorithm/classifier, and C is a regularization term, which determines how much the classification errors matter *(relative to the maximization of the margin)*.

Now, the function that the SVM minimizes to find the boundary will be:

$$\min_{w} ||w||^2 + C \sum_{i=1}^{N} max\left(0, 1 - y_i(w^T x_i + b)\right)$$

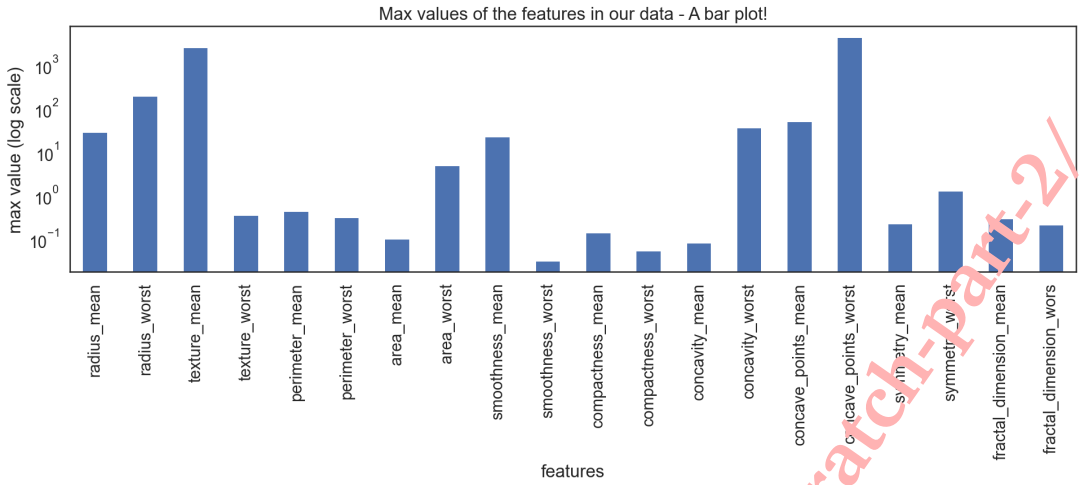
A small value of C creates a wider margin because errors will matter less. A large C creates a tighter margin because errors matter more. An infinite C parameter is a "hard" margin, which always minimizes error over the size of the boundary.

Support Vector Machines

(Hands-0m with real dataset)

– End-to-end project implementation and finding the best parameters –

Author: Dr. Junaid Qazi, PhD



From the above bar plot, we can see that there is significant variation in the range of the features. Some values are sufficiently larger than others. We know the importance of feature scaling and have seen the improvements in KNN lecture.

Let's get the scaled features and re-train our SVM model. (Code reference: KNN lecture)

Go to: L16: Support Vector Machines (SVMs) – Hands-on

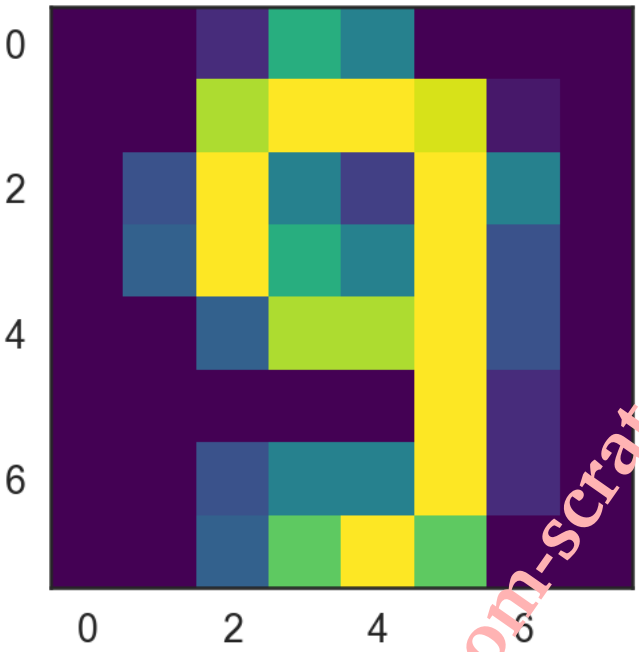
I am going to put couple of steps in a single cell, must be easier for you at this stage.

[48]:
```
# Importing StandardScaler and joblib, you can use pickle as well
from sklearn.preprocessing import StandardScaler
import joblib

# Creating instance 'scaler'
scaler = StandardScaler()

# fitting on features
scaler.fit(features) # our features are in features

# Saving the transformation, a good ML practice
joblib.dump(scaler, 'scaling_transformation.pkl')
print('transformation saved as scaling_transformation.pkl')

# Loading saved transformation
scaler = joblib.load('scaling_transformation.pkl')
print('Saved transformation is loaded.')

# transforming features
scaled_features = scaler.transform(features)
print('scaled features are in scaled_features')
```

```
transformation saved as scaling_transformation.pkl
Saved transformation is loaded.
scaled features are in scaled_features
```

(L16-4-6) Model re-training and evaluation using scaled features

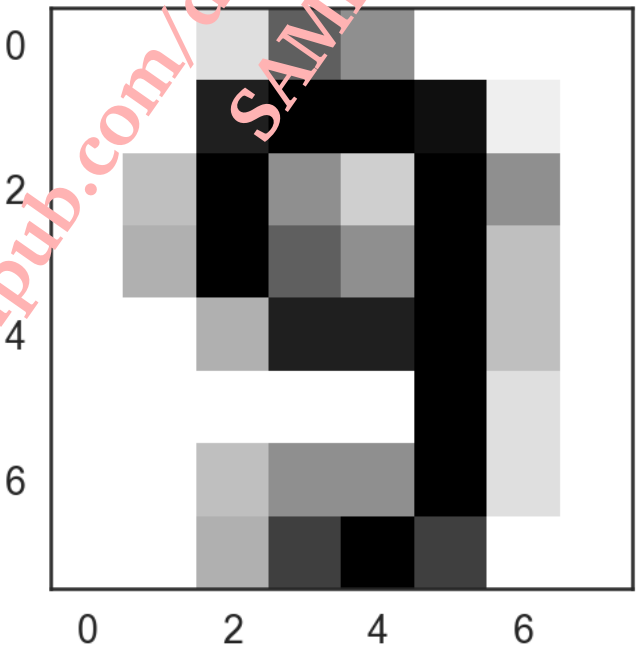
Let's split the data, train the model, get the predictions and print the confusion matrix, all in one cell of code.



If you notice the `mnist.keys()`, we have a key `'images'` in the list which contains the `8x8` matrices of all the digits, we can use them to view the numbers as well.

Let's try the digit on same index as above from images in gray scale.

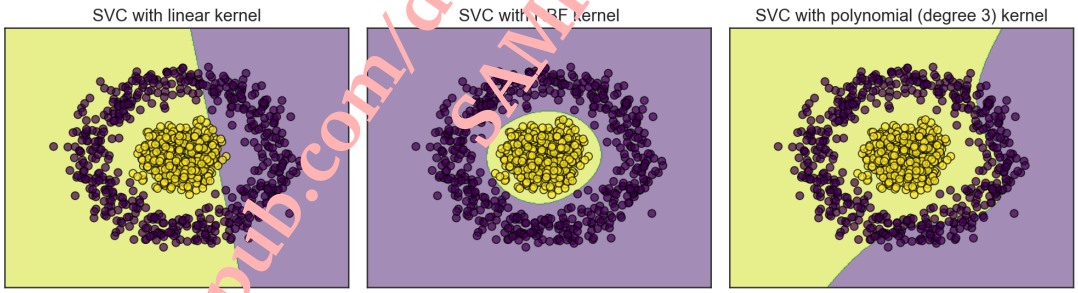
[7]:
```
# The same number as above
plt.imshow(mnist.images[index], cmap=plt.cm.gray_r)#, interpolation='nearest')
plt.show()
```



Go to: L17: SVMs and Logistic Regression – practice and comparisons

```
print("\twith linear kernel\t:",linear_svc.score(circles_X, circles_y))
print("\twith RBF kernel\t\t:",rbf_svc.score(circles_X, circles_y))
print("\twith polynomial kernel\t:",poly_svc.score(circles_X, circles_y))

# Creating a mesh for the plots
x_min, x_max = circles_X[:, 0].min() - 1, circles_X[:, 0].max() + 1
y_min, y_max = circles_X[:, 1].min() - 1, circles_X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, .02),
                     np.arange(y_min, y_max, .02))

# title for the plots
titles = ['SVC with linear kernel',
          'SVC with RBF kernel',
          'SVC with polynomial (degree 3) kernel']

plt.figure(figsize=(14,4))
for i, clf in enumerate((linear_svc, rbf_svc, poly_svc)):
    plt.subplot(1, 3, i + 1)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    # Adding result on a color plot
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.viridis, alpha=0.5)
    # Putting the training data
    plt.scatter(circles_X[:, 0],circles_X[:, 1], c=circles_y,
                s=50,edgecolors='black',cmap=plt.cm.viridis,alpha=0.7)
    plt.xticks(());plt.yticks(());plt.title(titles[i])
plt.tight_layout()
```

```
SVM scores are:
        with linear kernel      : 0.669
        with RBF kernel         : 1.0
        with polynomial kernel  : 0.418
```



Go to: L17: SVMs and Logistic Regression – practice and comparisons

Remember, we did not do hyperparameter tuning for iris and circles datasets and used the default parameters.

Think about some complex dataset for classification problem and compare the results from all the models that we have learned so far. Practice is a key for understanding. UCI Machine Learning Repository could be helpful to find the datasets.