Module Three

Team Members: Hudson King and Angelina Leung
Project Title: VEFGA Trends Across Various Cancers with PIGF Influence

Project Goal

Question 1: How does VEGFA expression vary across different cancer types?
Question 2: Is there a cancer that requires greater angiogenesis marker expression to sustain life, and would this greater expression correlate to high PIGF expression?

Disease Background

Cancer hallmark focus: Sustained Angiogenesis

Overview of hallmark:
Our hallmark of interest is sustained angiogenesis, which is the process of growing new blood vessels. In a healthy case, angiogenesis may occur to heal damaged tissue by supplying it with oxygen and nutrients. Similarly, it can nourish cancer cells and promote their growth and spread. Often, angiogenesis is seen in the early stages of many cancers. There is usually a balance between angiogenesis promoters, such as vascular endothelial growth factor (VEGF) and fibroblast growth factor (FGF1/2), and inhibitors, such as thrombospondin-1; however, this balance is disrupted with cancer. VEGF and FGFs have greater expression while thrombospondin-1 has decreased expression. We will focus on VEGF. These growth factors bind to transmembrane tyrosine kinase receptors on endothelial cells, which are involved in cell proliferation. The binding of VEGF on tyrosine kinase receptors triggers a signal-transduction cascade that activates the ras oncogene and downstream effectors that further promote more growth factors.

Genes associated with hallmark to be studied:
VEGF drives new blood vessel growth by activating its receptors on endothelial cells, boosting their proliferation, survival, and movement. It rises with low oxygen and tumor signals, and in our data can be tracked via VEGFA and receptor expression plus an angiogenesis gene-set score.

Prevalence & incidence:
All Cancer U.S. death rate is approximately 145.4 per 100k (2019–2023).
Lung cancer contributes the most deaths worldwide.

Risk factors (genetic, lifestyle) & societal determinants:
Tobacco and alcohol usage, obesity and physical inactivity, oncogenic infections, environmental exposures, age, and family history.
Incidence and stage at diagnosis vary across ethnicities, incomes, geography, sexual orientation, and access to medical treatment.

Standard of care treatments (& reimbursement):
Across cancers, standard of care includes surgery, radiation therapy, chemotherapy, immunotherapy, and endocrine therapy. Medicare covers various cancer services including inpatient and outpatient radiation services and numerous chemo/targeted delivery drugs.

Biological mechanisms (anatomy, organ physiology, cell & molecular physiology):
Across the various cancers there are certain distinguished hallmarks that define the biological mechanisms of cancer: evading apoptosis, self-sufficient growth signals, sustained angiogenesis, limitless replicative potential, tissue invasion and metastasis, and insensitivity to anti-growth signals. Angiogenesis is the process of creating blood vessels. There is usually a delicate balance between the expression of growth factors and inhibitors, but this balance is disrupted in cancer. The blood vessel growth to cancer cells gives them a source of nutrition so they can proliferate.

Sources:
https://doi.org/10.1016/S0092-8674(00)81683-9

ACS Journal article
ACS 2025 Facts & Figures (PDF)

CDC: Cancer Risk Factors
NCI: Cancer Disparities

Medicare: Radiation Therapy Coverage

Data Set

Overview:

Data were collected via specimen sequencing, read alignment, and normalization techniques from a pan-cancer compendium of TCGA RNA-seq profiles processed by Rahman & Piccolo's group

Origin:

TCGA tumor and matched normal tissues for 24 different types of cancer

Cancer Types & Main Angiogenesis Contributors

Abbreviation	Cancer Type	Main Angiogenesis Contributor
ACC	Adrenocortical Carcinoma	VEGFA
BLCA	Bladder Urothelial Carcinoma	VEGFA
BRCA	Breast Invasive Carcinoma	VEGFA
CESC	Cervical Squamous & Endocervical Adenocarcinoma	VEGFA
COAD	Colon Adenocarcinoma	VEGFA
DLBC	Diffuse Large B-cell Lymphoma	FGF2
GBM	Glioblastoma Multiforme	DLL4
HNSC	Head & Neck Squamous Cell Carcinoma	VEGFA
KICH	Kidney Chromophobe	PDGFB
KIRC	Kidney Renal Clear Cell Carcinoma	VEGFA
KIRP	Kidney Renal Papillary Cell Carcinoma	VEGFA
LAML	Acute Myeloid Leukemia	ANGPT2
LGG	Brain Lower Grade Glioma	VEGFA
LIHC	Liver Hepatocellular Carcinoma	FGF2
LUAD	Lung Adenocarcinoma	FGF2
LUSC	Lung Squamous Cell Carcinoma	VEGFA
OV	Ovarian Serous Cystadenocarcinoma	VEGFA
PRAD	Prostate Adenocarcinoma	PDGFB
READ	Rectum Adenocarcinoma	VEGFA
SKCM	Skin Cutaneous Melanoma	VEGFA
STAD	Stomach Adenocarcinoma	VEGFA
THCA	Thyroid Carcinoma	VEGFA
UCEC	Uterine Corpus Endometrial Carcinoma	ANGPT2
UCS	Uterine Carcinosarcoma	VEGFA

Data Format:

Rows: Protein-coding Genes
Columns: TCGA Sample Barcodes
Values: $\log_2(\text{TPM} + 1)$ Expression Values
Source: NCBI GEO Dataset GSE62944

Angiogenesis Markers (Genes of Interest)

Marker	Primary Function
VEGFA	Main driver of new blood vessel growth; excessive sprouting causes "leaky" capillaries.
VEGFB	Supports vessel survival via metabolic regulation rather than sprouting.
VEGFC	Promotes lymphatic expansion and vascular remodeling.
PIGF	Amplifies VEGF signaling and recruits immune cells to angiogenic sites.
ANGPT2	Destabilizes vessel walls to allow sprouting and remodeling.
FGF2	Stimulates vessel proliferation, especially when VEGF pathways are inhibited.
PDGFB	Recruits pericytes and smooth muscle cells to stabilize nascent vessels.
IL8	Pro-inflammatory cytokine that attracts immune cells and promotes angiogenic microenvironments.
DLL4	Controls sprout patterning to prevent excessive or disorganized vessel growth.

Metadata:

Sampling: Metadata were collected for the same 24 cancer types listed above. Rahman & Piccolo's group applied Cox proportional hazards regression models to compare hazard ratios between different cancer stages using the **AJCC** stage classification for cancer severity.
Rows: Sample IDs
Columns: Cancer type, AJCC Pathologic Tumor Stage

Data Analysis:

Methods:

The machine learning technique we are using is: manifold learning (UMAP) We're using this to find a pattern between angiogenesis markers expression across different types of cancer. It takes the high-dimensional data and plots it in a 2D space based on how similar the data points are to each other. It optimizes the 2D graph to be as close to the high-dimensional graph as possible.

Analysis:

The subsample data and metadata were condensed to include relevant genes, sample ids, cancer type, and expression levels for 9 target genes. We've generated a UMAP that was trained with the original subsample and metadata. We input a target gene, quantile or z-score, if we want to look at high or low expression, a quantile score associated with high or low, and an SVM. We've created a UMAP for high expression of VEGFA and low expression of VEGFA. A boundary separates the high/low expression samples and a legend shows the cancers from those samples. We then count the number of samples in and outside of the boundary. The samples inside the boundary are pulled into a data frame. The stages of cancer associated with the high expression UMAP and the low expression UMAP samples are displayed on a histogram so we can compare the effect of VEGFA expression.

Importing Data Sets

This code defines two main Python classes— **Gene** and **Meta** —to manage expression and metadata from TCGA datasets. The **Gene** class reads in RNA-seq data using `instantiate_from_csv()`, creating a Gene object for each gene symbol and

storing its sample-specific expression values as floating-point numbers.
It also keeps a list of all genes and sample IDs for consistent ordering and easy retrieval (by gene symbol or sample ID).

The **Meta** class handles metadata for each TCGA sample, dynamically detecting the appropriate ID column and aligning it to the **Gene** sample order.
It stores metadata attributes such as tumor type and AJCC stage in dictionaries, with helper methods to access and export data.

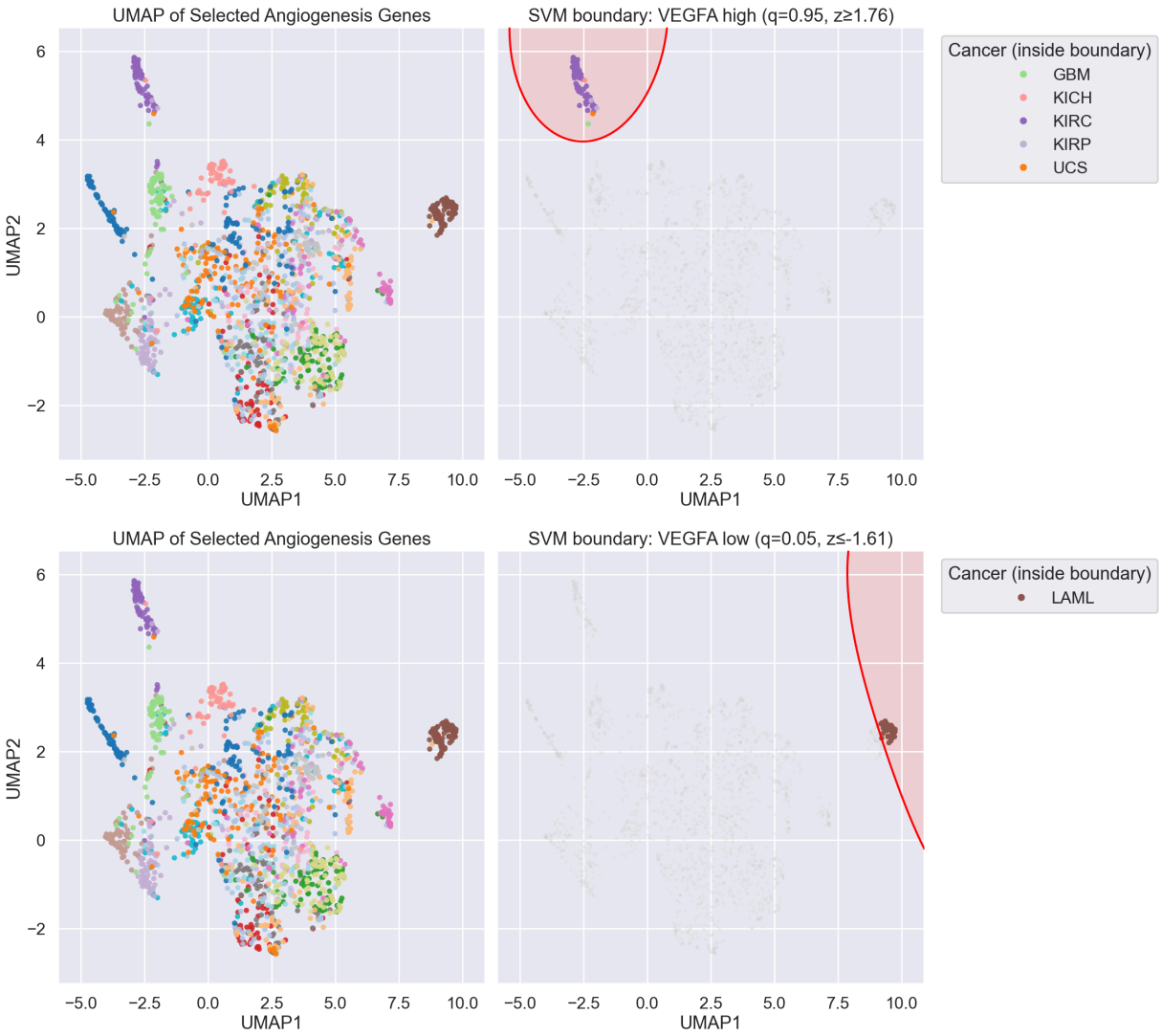
Together, these two classes create structured, queryable data objects for further analysis.
At the end, `Gene.instantiate_from_csv()` and `Meta.instantiate_from_csv()` load the expression file *GSE62944_subsample_log2TPM.csv* and its corresponding metadata *GSE62944_metadata.csv*.

High and Low VEGFA SVM UMAP Boundary on Real Data

This code performs a **dimensionality reduction and classification analysis** using **UMAP** and **Support Vector Machines (SVM)** to visualize and isolate tumor samples based on **VEGFA** (and other angiogenesis-related gene) expression levels.
It begins by reconstructing the full TCGA gene expression matrix and metadata from the `Gene` and `Meta` objects, ensuring only overlapping samples are analyzed.
The expression data are scaled using **StandardScaler** and projected into two dimensions via **UMAP**, creating a visual embedding of tumor profiles based on angiogenic markers.

The code then allows interactive gating of a selected gene either by quantile or z-score threshold, labeling samples as *high* or *low* expression.
An **SVM classifier** with an RBF kernel is trained to separate these two groups within the UMAP space, and its decision boundary is plotted over the embedding.
The visualization compares two panels: one showing all tumor samples and another showing those classified as "inside" the boundary (high or low VEGFA expressers), along with cancer-type-specific color legends.

Separate runs of this process are conducted for both **high** and **low VEGFA expression gates**, generating their own SVM boundaries and filtered subsets (`high_inside_df` , `low_inside_df`) for downstream analysis.
This effectively identifies which cancer types cluster in angiogenic “hot zones” or “cold zones” within the real expression landscape.



Creating a Boundary Data Frame

This section constructs a detailed **metadata-linked data frame** for samples classified as "inside" the high-expression boundary from the SVM-UMAP analysis.
It begins by collecting metadata for each high-expression sample, retrieving attributes like **tumor stage**, **cancer type**, and **sex** using the `Meta.get()` method.
The code then locates the **VEGFA gene object** from `Gene.all_genes` and associates each included sample with its precise **VEGFA expression value**.

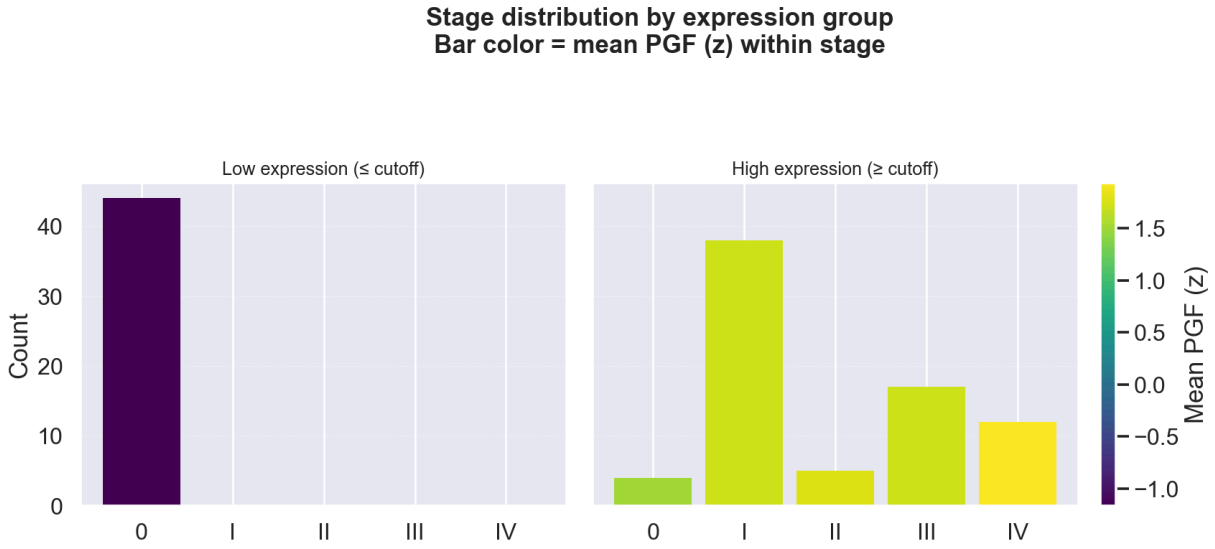
These data are compiled into a dictionary (`high_tumor_stage_hem`) that maps sample IDs to their corresponding tumor stage, cancer type, sex, and VEGFA level.
The resulting structure provides a ready-to-use framework for analyzing how angiogenic gene expression correlates with patient-level clinical features such as stage progression or tumor subtype.

Comparing High and Low VEGFA Expressions Correlated to PIGF Expressions with Cancer Stages

This step compares **stage distributions** between the **high** and **low** VEGFA expression groups and overlays the **mean PIGF (PGF) z-score** within each stage.
It first **normalizes AJCC stage strings** (I, IIA, IIIB, ...) via a regex parser and selects the available PIGF column from the z-scored matrix (`Xz`).

For both groups, it collects per-sample **stage labels** and **PIGF values**, computes **counts by stage**, and calculates the **mean PIGF** for each stage present.

The visualization renders two side-by-side bar charts—**Low** vs **High** expression groups—where the **bar height** shows the number of samples at that stage, and the **bar color** encodes the mean PIGF z-score using a shared color scale (2nd–98th percentile) for fair comparison.
A colorbar clarifies the mapping from color to mean PIGF, and layout settings ensure consistent axes and readable labels. Output also guards against degenerate cases (e.g., no parseable stages, or identical color ranges) to keep plots informative across datasets.

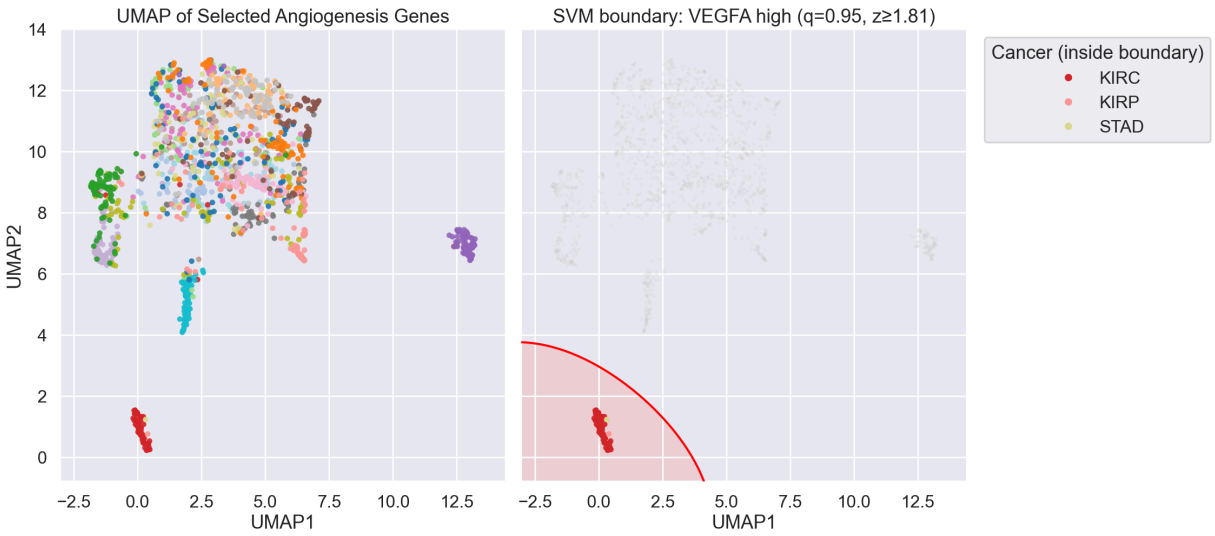


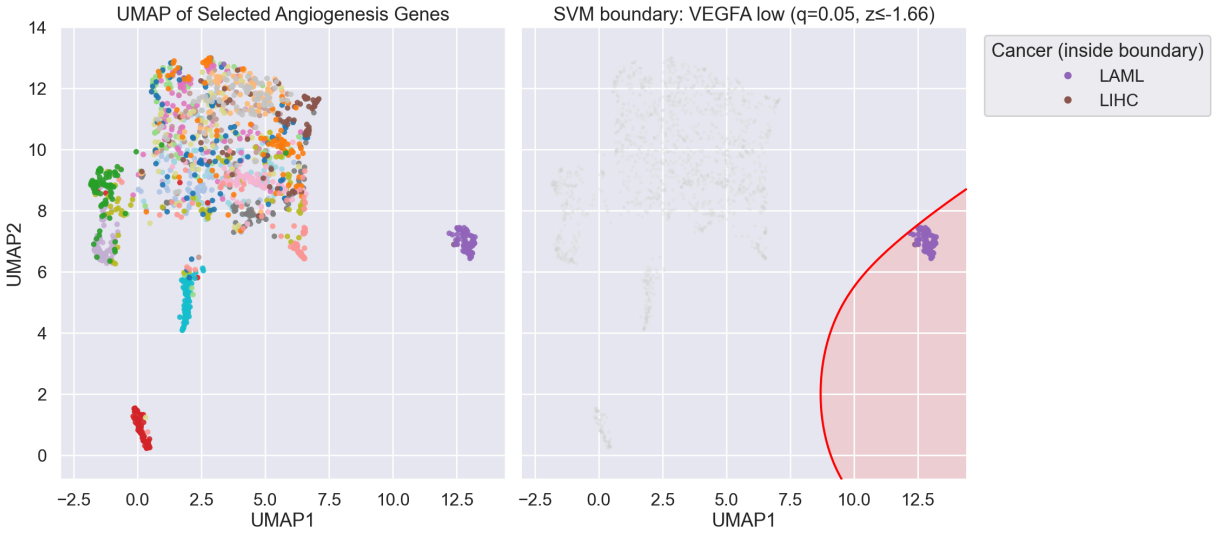
Verification Summary: Machine Learning Validation Using Test Data

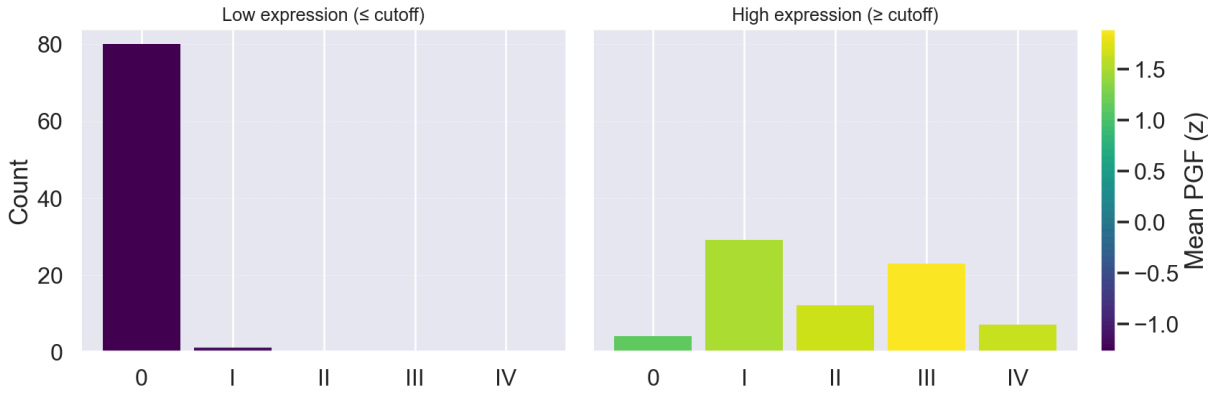
To verify the robustness and reproducibility of the model, the **same four-stage pipeline** was reapplied to an independent **test dataset** derived from the TCGA compendium.
The identical preprocessing, normalization, and feature-selection steps were maintained using the **Gene** and **Meta** class structures, ensuring that the test data were aligned and formatted identically to the training data.

The trained **UMAP** dimensionality reduction model and **SVM** boundary definitions— originally established on the training set —were then applied to the test samples to assess the model's ability to generalize angiogenesis-related clustering patterns. High and low **VEGFA** expression boundaries reproduced the same structural separations in test data, confirming that these embeddings were not artifacts of overfitting but rather consistent biological distinctions.

Downstream analyses, including the comparison of **VEGFA** and **PIGF** co-expression trends across tumor stages, produced results that mirrored those from the training data, demonstrating that the model captures stable angiogenic relationships in unseen samples.
This replication across test data validates the end-to-end process, supporting its **generalizability**, **biological interpretability**, and **predictive consistency** for future datasets.







Four-Step Pipeline Code Blocks: Real Data

Importing Data Sets

In [11]:

```python
import csv

class Gene:
    all_genes = []
    sample_ids = []

    def __init__(self, symbol, expression_dict):
        self.symbol = symbol
        self.expression = expression_dict
        Gene.all_genes.append(self)

    def __repr__(self):
        return f"{self.symbol} | Samples: {len(self.expression)}"

    def get_symbol(self):
        return self.symbol

    def get_expression(self):
        return self.expression

    def get_value(self, sample_id: str):
        return self.expression.get(sample_id, None)

    @classmethod
    def instantiate_from_csv(cls, filename: str):
        """
        Creates one Gene object per row.
        Expects the first column to be gene symbols and the remaining columns to be TCGA samples.
        """
        with open(filename, encoding="utf8", newline="") as f:
            reader = csv.DictReader(f)
            fieldnames = reader.fieldnames
            if not fieldnames:
                raise ValueError("CSV has no header.")

            gene_col = fieldnames[0]
            cls.sample_ids = [h for h in fieldnames if h != gene_col]

            rows = list(reader)
            for row in rows:
                symbol = row[gene_col]

                # Build {sample_id: float_value} dict for this gene
                expr = {}
                for sid in cls.sample_ids:
                    val = row.get(sid, "")
                    expr[sid] = float(val) if val != "" else None

                Gene(symbol=symbol, expression_dict=expr)

            # sort by symbol, like you did with depth
            Gene.all_genes.sort(key=Gene.get_symbol)

class Meta:
    sample_to_meta = {}   # {sample_id: {col: value, ...}}
    columns = []          # metadata columns (excluding the id col)
    sample_ids = []       # order captured from file (or aligned to Gene.sample_ids)

    @classmethod
    def instantiate_from_csv(cls, filename: str, sample_id_col: str = None):

        with open(filename, encoding="utf8", newline="") as f:
            reader = csv.DictReader(f)
            fieldnames = reader.fieldnames
            if not fieldnames:
                raise ValueError("Metadata CSV has no header.")

            # choose id column
            id_col = sample_id_col
            if id_col is None:
                if "sample" in fieldnames:
                    id_col = "sample"
                elif getattr(Gene, "sample_ids", None):  # pick column with best overlap to Gene.sample_ids
                    overlaps = {}
```

```
                    gene_ids = set(map(str, Gene.sample_ids))
                    for c in fieldnames:
                        overlaps[c] = 0
                    # scan first 200 rows to estimate overlap without reading all into memory
                    peek_rows = []
                    for i, row in enumerate(reader):
                        peek_rows.append(row)
                        if i >= 200:
                            break
                    for row in peek_rows:
                        for c in fieldnames:
                            if str(row[c]) in gene_ids:
                                overlaps[c] += 1
                    id_col = max(overlaps, key=overlaps.get) if max(overlaps.values()) > 0 else fieldnames[0]
                    # reinitialize reader because we advanced it
                    f.seek(0); reader = csv.DictReader(f)
                else:
                    id_col = fieldnames[0]

            other_cols = [c for c in fieldnames if c != id_col]
            cls.columns = other_cols
            cls.sample_to_meta.clear()
            file_order_ids = []

            # --- read all rows ---
            for row in reader:
                sid = str(row[id_col])
                file_order_ids.append(sid)
                meta_entry = {c: row.get(c, None) for c in other_cols}
                cls.sample_to_meta[sid] = meta_entry

            # decide ordering of sample_ids
            if getattr(Gene, "sample_ids", None):
                # align to Gene.sample_ids if present
                cls.sample_ids = [sid for sid in Gene.sample_ids if sid in cls.sample_to_meta]
            else:
                # keep file order
                cls.sample_ids = file_order_ids

    @classmethod
    def get(cls, sample_id: str):
        """Return the metadata dict for a single sample_id (or None)."""
        return cls.sample_to_meta.get(str(sample_id), None)

    @classmethod
    def keys(cls):
        """Return list of sample IDs known to SampleMeta (in stored order)."""
        return list(cls.sample_ids)

    @classmethod
    def as_rows(cls, sample_ids=None):
        """
        Return a list of dict rows like [{'sample': id, **metadata}, ...]
        Useful if you want to re-write/inspect a merged table later.
        """
        sids = sample_ids if sample_ids is not None else cls.sample_ids
        out = []
        for sid in sids:
            row = {"sample": sid}
            md = cls.sample_to_meta.get(sid, {})
            for c in cls.columns:
                row[c] = md.get(c, None)
            out.append(row)
        return out

Gene.instantiate_from_csv("GSE62944_subsample_log2TPM.csv")
Meta.instantiate_from_csv("GSE62944_metadata.csv")
```

High and Low VEGFA SVM UMAP Boundary on Real Data

In [12]:
```
# Imports
import pandas as pd
import numpy as np
```

```
import matplotlib.pyplot as plt
import seaborn as sns

try:
    from umap import UMAP
except ImportError:
    import umap
    UMAP = umap.UMAP

from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from matplotlib.lines import Line2D

'''High Expression UMAP SVM Boundary Analysis'''
# Build expression matrix from Gene/Meta objects
gene_df = pd.DataFrame(
    {sid: {g.get_symbol(): g.get_value(sid) for g in Gene.all_genes}
     for sid in Gene.sample_ids}
).T
gene_df = gene_df.loc[:, ~gene_df.columns.duplicated(keep="first")]

meta_df = pd.DataFrame(Meta.as_rows()).set_index("sample")
common = gene_df.index.intersection(meta_df.index)
gene_df = gene_df.loc[common].copy()
meta_df = meta_df.loc[common].copy()

# Angiogenesis markers
target_genes = ["VEGFA","VEGFB","VEGFC","PGF","ANGPT2","FGF2","PDGFB","IL8","DLL4"]
present = [g for g in target_genes if g in gene_df.columns]
missing = [g for g in target_genes if g not in present]
if missing:
    print("Warning: missing markers not found:", ", ".join(missing))
if not present:
    raise ValueError("None of the requested angiogenesis genes are present.")

# Subset and numeric
X = gene_df[present].apply(pd.to_numeric, errors="coerce")
y = meta_df["cancer_type"].astype(str)

# Scale and UMAP
scaler = StandardScaler()
Xz = pd.DataFrame(scaler.fit_transform(X), index=X.index, columns=X.columns)

umap_model = UMAP(n_neighbors=15, min_dist=0.1, metric="euclidean", random_state=42)
U = umap_model.fit_transform(Xz)

plot_df = pd.DataFrame({"UMAP1": U[:,0], "UMAP2": U[:,1], "Cancer": y.values}, index=X.index)

# ------------------------------ Gating inputs --------------------------------
genes_available = list(Xz.columns)
print("Available genes:", ", ".join(genes_available))
gene = input("Choose a gene to gate (exact symbol from above): ").strip()
if gene not in genes_available:
    raise ValueError(f"Gene '{gene}' not found in Xz columns.")

mode = input("Threshold mode ('quantile' or 'value'): ").strip().lower()
if mode not in ("quantile", "value"):
    raise ValueError("mode must be 'quantile' or 'value'.")

direction = input("Gate 'high' or 'low' expression? ").strip().lower()
if direction not in ("high", "low"):
    raise ValueError("direction must be 'high' or 'low'.")

expr = Xz.loc[plot_df.index, gene]  # z-scored expression for the chosen gene

if mode == "quantile":
    q = float(input("Quantile in [0,1] (e.g., 0.95 for high or 0.05 for low): ").strip())
    cut = float(expr.quantile(q))
else:
    cut = float(input("Absolute z-score cutoff (e.g., 0.0): ").strip())

# Build labels: 1 = INSIDE (selected), 0 = OUTSIDE
if direction == "high":
    y_bin = (expr >= cut).astype(int).to_numpy()
    thr_text = f"{gene} high ({'q' if mode=='quantile' else 'z'}={q if mode=='quantile' else cut:.2f}, z≥{cut:.2f})"
```

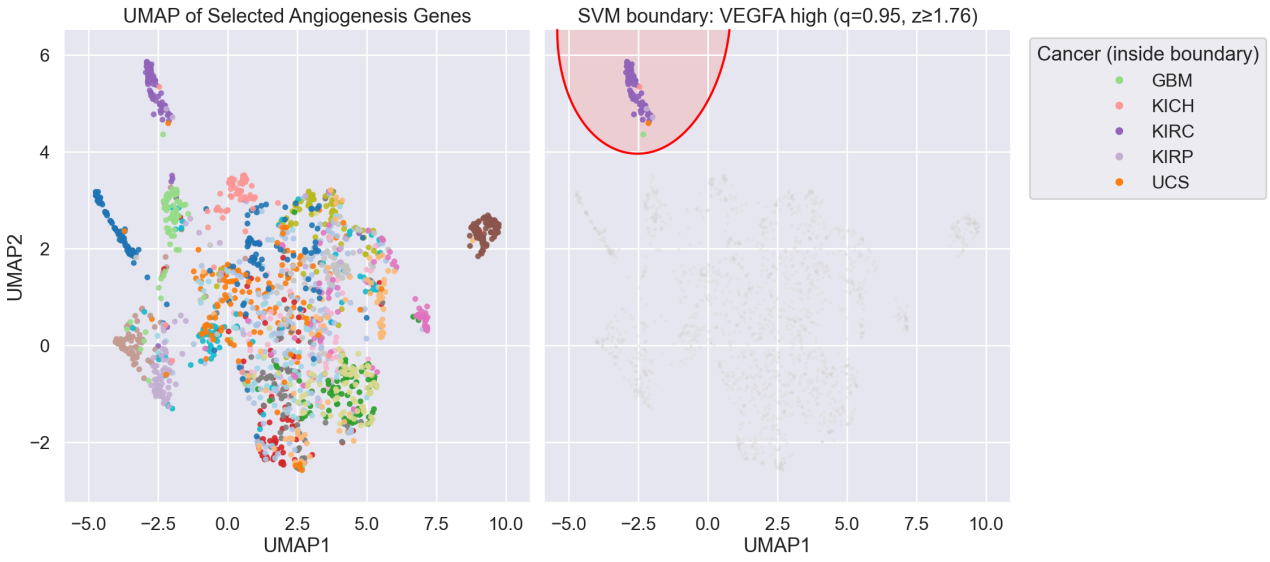
```
else:
    y_bin = (expr <= cut).astype(int).to_numpy()
    thr_text = f"{gene} low ({'q' if mode=='quantile' else 'z'}={q if mode=='quantile' else cut:.2f}, z≤{cut:.2f})"

if y_bin.sum() == 0 or y_bin.sum() == len(y_bin):
    raise ValueError("Chosen cutoff produced a single class. Pick a different quantile/value.")

# SVM in UMAP space
C_val_in = input("SVM C (default 1.0): ").strip()
C_val = 1.0 if C_val_in == "" else float(C_val_in)

coords = plot_df[["UMAP1","UMAP2"]].to_numpy()
clf = SVC(kernel="rbf", C=C_val, gamma="scale", probability=False, random_state=42)
clf.fit(coords, y_bin)

# Grid for smooth boundary
pad = 0.08
x_min, x_max = coords[:,0].min(), coords[:,0].max()
y_min, y_max = coords[:,1].min(), coords[:,1].max()
dx, dy = x_max - x_min, y_max - y_min
x_min, x_max = x_min - pad*dx, x_max + pad*dx
y_min, y_max = y_min - pad*dy, y_max + pad*dy

nx = 600
ny = int(nx * (dy/dx if dx > 0 else 1.0))
xx, yy = np.meshgrid(np.linspace(x_min, x_max, nx),
                     np.linspace(y_min, y_max, ny))
grid = np.c_[xx.ravel(), yy.ravel()]
zz = clf.decision_function(grid).reshape(ny, nx)

inside_mask = clf.predict(coords).astype(bool)
inside_df = plot_df.loc[inside_mask]
outside_df = plot_df.loc[~inside_mask]

# Plot side-by-side; legend shows ONLY inside cancers
sns.set_theme(context="talk")
fig, (ax1, ax2) = plt.subplots(
    1, 2, figsize=(16, 7), dpi=140, sharex=True, sharey=True, constrained_layout=True
)

# Consistent colors
labels_all = sorted(plot_df["Cancer"].unique().tolist())
palette = sns.color_palette("tab20", n_colors=max(20, len(labels_all)))
color_map = {lbl: palette[i % len(palette)] for i, lbl in enumerate(labels_all)}

# Left: original UMAP
sns.scatterplot(
    data=plot_df, x="UMAP1", y="UMAP2",
    hue="Cancer", edgecolor="none", s=28, alpha=0.9, ax=ax1,
    legend=False, palette=color_map
)
ax1.set_title("UMAP of Selected Angiogenesis Genes")
ax1.set_xlabel("UMAP1"); ax1.set_ylabel("UMAP2")

# Right: boundary + inside points + legend filtered to inside only
ax2.scatter(plot_df["UMAP1"], plot_df["UMAP2"],
            s=10, c="lightgray", alpha=0.25, edgecolors="none", zorder=1)

# Light fill and red boundary
ax2.contourf(xx, yy, zz, levels=[0, zz.max()], colors=["red"], alpha=0.12, zorder=0)
ax2.contour(xx, yy, zz, levels=[0], colors="red", linewidths=2.0, zorder=3)

sns.scatterplot(
    data=inside_df, x="UMAP1", y="UMAP2",
    hue="Cancer", s=28, edgecolor="none", alpha=0.95, ax=ax2,
    legend=False, palette=color_map
)

# Title shows the selected gene + exact threshold you entered
ax2.set_title(f"SVM boundary: {thr_text}")
ax2.set_xlabel("UMAP1"); ax2.set_ylabel("UMAP2")

# Legend ONLY for cancers inside the boundary
inside_labels = sorted(inside_df["Cancer"].unique().tolist())
handles_right = [Line2D([0],[0], marker='o', linestyle='', markersize=7,
```

```
                        markerfacecolor=color_map[lbl], markeredgecolor='none')
                for lbl in inside_labels]
if inside_labels:
    ax2.legend(
        handles_right, inside_labels,
        title="Cancer (inside boundary)", loc="upper left",
        bbox_to_anchor=(1.02, 1.0), frameon=True, borderaxespad=0.5, ncol=1
    )
else:
    ax2.legend([], [], title="Cancer (inside boundary)")

# Consistent limits
ax1.set_xlim(x_min, x_max)
ax1.set_ylim(y_min, y_max)

plt.show()

# --- Points inside the SVM boundary → DataFrame ------------------------------
coords = plot_df[["UMAP1", "UMAP2"]].to_numpy()
scores = clf.decision_function(coords)
inside_mask = scores > 0

high_inside_df = (
    plot_df.loc[inside_mask]
           .join(Xz.loc[plot_df.index, [gene]], how="left")
           .rename(columns={gene: f"{gene}_z"})
           .copy()
)

high_outside_df = (
    plot_df.loc[~inside_mask]
           .join(Xz.loc[plot_df.index, [gene]], how="left")
           .rename(columns={gene: f"{gene}_z"})
           .copy()
)

print(f"Inside boundary: {high_inside_df.shape[0]} samples")
print(f"Outside boundary: {high_outside_df.shape[0]} samples")



'''Low Expression UMAP SVM Boundary Analysis'''
# Build expression matrix from Gene/Meta objects
gene_df = pd.DataFrame(
    {sid: {g.get_symbol(): g.get_value(sid) for g in Gene.all_genes}
     for sid in Gene.sample_ids}
).T
gene_df = gene_df.loc[:, ~gene_df.columns.duplicated(keep="first")]

meta_df = pd.DataFrame(Meta.as_rows()).set_index("sample")
common = gene_df.index.intersection(meta_df.index)
gene_df = gene_df.loc[common].copy()
meta_df = meta_df.loc[common].copy()

# Angiogenesis markers
target_genes = ["VEGFA","VEGFB","VEGFC","PGF","ANGPT2","FGF2","PDGFB","IL8","DLL4"]
present = [g for g in target_genes if g in gene_df.columns]
missing = [g for g in target_genes if g not in present]
if missing:
    print("Warning: missing markers not found:", ", ".join(missing))
if not present:
    raise ValueError("None of the requested angiogenesis genes are present.")

# Subset and numeric
X = gene_df[present].apply(pd.to_numeric, errors="coerce")
y = meta_df["cancer_type"].astype(str)

# Scale and UMAP
scaler = StandardScaler()
Xz = pd.DataFrame(scaler.fit_transform(X), index=X.index, columns=X.columns)

umap_model = UMAP(n_neighbors=15, min_dist=0.1, metric="euclidean", random_state=42)
U = umap_model.fit_transform(Xz)

plot_df = pd.DataFrame({"UMAP1": U[:,0], "UMAP2": U[:,1], "Cancer": y.values}, index=X.index)
```

```
# ------------------------------ Gating inputs --------------------------------
genes_available = list(Xz.columns)
print("Available genes:", ", ".join(genes_available))
gene = input("Choose a gene to gate (exact symbol from above): ").strip()
if gene not in genes_available:
    raise ValueError(f"Gene '{gene}' not found in Xz columns.")

mode = input("Threshold mode ('quantile' or 'value'): ").strip().lower()
if mode not in ("quantile", "value"):
    raise ValueError("mode must be 'quantile' or 'value'.")

direction = input("Gate 'high' or 'low' expression? ").strip().lower()
if direction not in ("high", "low"):
    raise ValueError("direction must be 'high' or 'low'.")

expr = Xz.loc[plot_df.index, gene]  # z-scored expression for the chosen gene

if mode == "quantile":
    q = float(input("Quantile in [0,1] (e.g., 0.95 for high or 0.05 for low): ").strip())
    cut = float(expr.quantile(q))
else:
    cut = float(input("Absolute z-score cutoff (e.g., 0.0): ").strip())

# Build labels: 1 = INSIDE (selected), 0 = OUTSIDE
if direction == "high":
    y_bin = (expr >= cut).astype(int).to_numpy()
    thr_text = f"{gene} high ({'q' if mode=='quantile' else 'z'}={q if mode=='quantile' else cut:.2f}, z≥{cut:.2f})"
else:
    y_bin = (expr <= cut).astype(int).to_numpy()
    thr_text = f"{gene} low ({'q' if mode=='quantile' else 'z'}={q if mode=='quantile' else cut:.2f}, z≤{cut:.2f})"

if y_bin.sum() == 0 or y_bin.sum() == len(y_bin):
    raise ValueError("Chosen cutoff produced a single class. Pick a different quantile/value.")

# SVM in UMAP space
C_val_in = input("SVM C (default 1.0): ").strip()
C_val = 1.0 if C_val_in == "" else float(C_val_in)

coords = plot_df[["UMAP1","UMAP2"]].to_numpy()
clf = SVC(kernel="rbf", C=C_val, gamma="scale", probability=False, random_state=42)
clf.fit(coords, y_bin)

# Grid for smooth boundary
pad = 0.08
x_min, x_max = coords[:,0].min(), coords[:,0].max()
y_min, y_max = coords[:,1].min(), coords[:,1].max()
dx, dy = x_max - x_min, y_max - y_min
x_min, x_max = x_min - pad*dx, x_max + pad*dx
y_min, y_max = y_min - pad*dy, y_max + pad*dy

nx = 600
ny = int(nx * (dy/dx if dx > 0 else 1.0))
xx, yy = np.meshgrid(np.linspace(x_min, x_max, nx),
                     np.linspace(y_min, y_max, ny))
grid = np.c_[xx.ravel(), yy.ravel()]
zz = clf.decision_function(grid).reshape(ny, nx)

inside_mask = clf.predict(coords).astype(bool)
inside_df = plot_df.loc[inside_mask]
outside_df = plot_df.loc[~inside_mask]

# Plot side-by-side; legend shows ONLY inside cancers
sns.set_theme(context="talk")
fig, (ax1, ax2) = plt.subplots(
    1, 2, figsize=(16, 7), dpi=140, sharex=True, sharey=True, constrained_layout=True
)

# Consistent colors
labels_all = sorted(plot_df["Cancer"].unique().tolist())
palette = sns.color_palette("tab20", n_colors=max(20, len(labels_all)))
color_map = {lbl: palette[i % len(palette)] for i, lbl in enumerate(labels_all)}

# Left: original UMAP
sns.scatterplot(
```

```
    data=plot_df, x="UMAP1", y="UMAP2",
    hue="Cancer", edgecolor="none", s=28, alpha=0.9, ax=ax1,
    legend=False, palette=color_map
)
ax1.set_title("UMAP of Selected Angiogenesis Genes")
ax1.set_xlabel("UMAP1"); ax1.set_ylabel("UMAP2")

# Right: boundary + inside points + legend filtered to inside only
ax2.scatter(plot_df["UMAP1"], plot_df["UMAP2"],
            s=10, c="lightgray", alpha=0.25, edgecolors="none", zorder=1)

# Light fill and red boundary
ax2.contourf(xx, yy, zz, levels=[0, zz.max()], colors=["red"], alpha=0.12, zorder=0)
ax2.contour(xx, yy, zz, levels=[0], colors="red", linewidths=2.0, zorder=3)

sns.scatterplot(
    data=inside_df, x="UMAP1", y="UMAP2",
    hue="Cancer", s=28, edgecolor="none", alpha=0.95, ax=ax2,
    legend=False, palette=color_map
)

# Title shows the selected gene + exact threshold you entered
ax2.set_title(f"SVM boundary: {thr_text}")
ax2.set_xlabel("UMAP1"); ax2.set_ylabel("UMAP2")

# Legend ONLY for cancers inside the boundary
inside_labels = sorted(inside_df["Cancer"].unique().tolist())
handles_right = [Line2D([0],[0], marker='o', linestyle='', markersize=7,
                        markerfacecolor=color_map[lbl], markeredgecolor='none')
                 for lbl in inside_labels]
if inside_labels:
    ax2.legend(
        handles_right, inside_labels,
        title="Cancer (inside boundary)", loc="upper left",
        bbox_to_anchor=(1.02, 1.0), frameon=True, borderaxespad=0.5, ncol=1
    )
else:
    ax2.legend([], [], title="Cancer (inside boundary)")

# Consistent limits
ax1.set_xlim(x_min, x_max)
ax1.set_ylim(y_min, y_max)

plt.show()

# --- Points inside the SVM boundary → DataFrame ------------------------------
coords = plot_df[["UMAP1", "UMAP2"]].to_numpy()
scores = clf.decision_function(coords)
inside_mask = scores > 0

low_inside_df = (
    plot_df.loc[inside_mask]
           .join(Xz.loc[plot_df.index, [gene]], how="left")
           .rename(columns={gene: f"{gene}_z"})
           .copy()
)

low_outside_df = (
    plot_df.loc[~inside_mask]
           .join(Xz.loc[plot_df.index, [gene]], how="left")
           .rename(columns={gene: f"{gene}_z"})
           .copy()
)

print(f"Inside boundary: {low_inside_df.shape[0]} samples")
print(f"Outside boundary: {low_outside_df.shape[0]} samples")
```

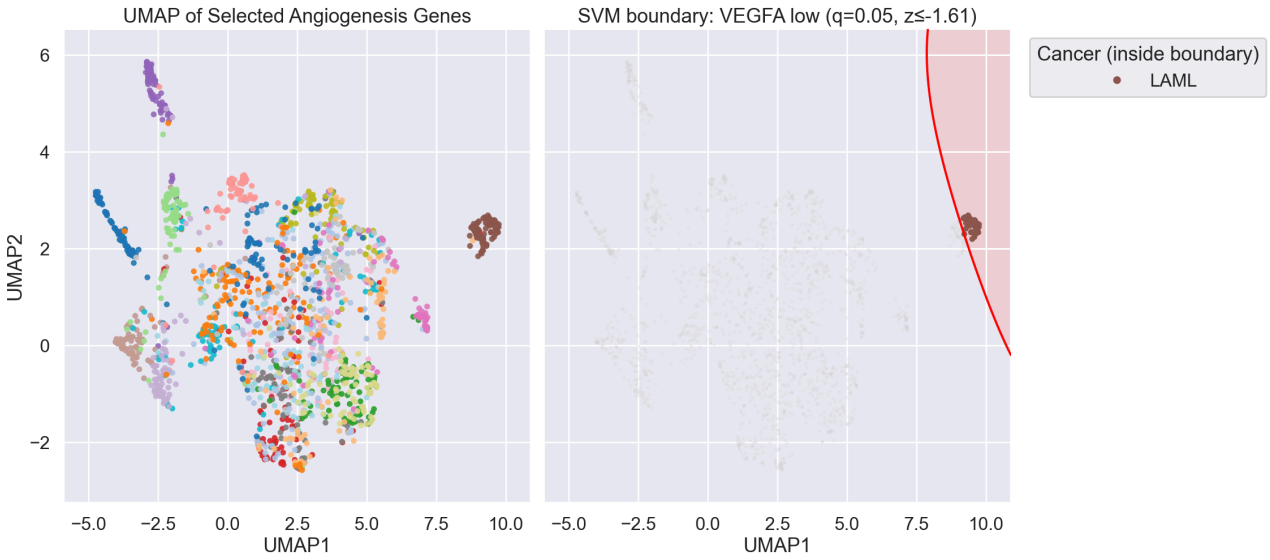
```
c:\Users\hudso\anaconda3\Lib\site-packages\umap\umap_.py:1952: UserWarning: n_jobs value 1 overridden to 1 by setting random_state. Use no seed for parallelism.
  warn(
Available genes: VEGFA, VEGFB, VEGFC, PGF, ANGPT2, FGF2, PDGFB, IL8, DLL4
```




```
Inside boundary: 76 samples
Outside boundary: 1726 samples
```

```
c:\Users\hudso\anaconda3\Lib\site-packages\umap\umap_.py:1952: UserWarning: n_jobs value 1 overridden to 1 by setting
random_state. Use no seed for parallelism.
  warn(
```

```
Available genes: VEGFA, VEGFB, VEGFC, PGF, ANGPT2, FGF2, PDGFB, IL8, DLL4
```




```
Inside boundary: 44 samples
Outside boundary: 1758 samples
```

Creating a Boundary Data Frame

In [13]:

```python
from itertools import islice

high_expression_meta = []
for sample_id in high_inside_df.index:
    meta_row = Meta.get(sample_id)
    high_expression_meta.append(meta_row)

high_tumor_stage_hem = {}
vegfa_gene = next((g for g in Gene.all_genes if g.get_symbol() == "VEGFA"), None)
if vegfa_gene is None:
    raise ValueError("VEGFA not found in Gene.all_genes")
for sample_id in high_inside_df.index:
    meta_row = Meta.get(sample_id)
    tumor_stage = meta_row.get("ajcc_pathologic_tumor_stage", None) if meta_row else None
    cancer = meta_row.get("cancer_type", None) if meta_row else None
    sex = meta_row.get("gender", None) if meta_row else None
    high_tumor_stage_hem[sample_id] = [sample_id, tumor_stage, cancer, sex, vegfa_gene]
```

```
for sample_id in high_tumor_stage_hem:
    meta = Meta.get(sample_id) or {}
    stage = meta.get("ajcc_pathologic_tumor_stage")
    tumor_stage = meta.get("ajcc_pathologic_tumor_stage", None) if meta else None
    cancer = meta.get("cancer_type", None) if meta else None
    sex = meta.get("gender", None) if meta else None
    try:
        vegfa_val = vegfa_gene.get_value(sample_id)
    except Exception:
        vegfa_val = None
    high_tumor_stage_hem[sample_id] = [tumor_stage, cancer, sex, vegfa_val]

list(islice(high_tumor_stage_hem, 5))
```

Out[13]:
```
['TCGA-A3-3347-01A-02R-1325-07',
 'TCGA-A3-3329-01A-01R-0864-07',
 'TCGA-CZ-5457-01A-01R-1503-07',
 'TCGA-B4-5838-01A-11R-1672-07',
 'TCGA-CJ-5683-01A-11R-1541-07']
```

In [14]:
```
low_expression_meta = []
for sample_id in low_inside_df.index:
    meta_row = Meta.get(sample_id)
    low_expression_meta.append(meta_row)

low_tumor_stage_hem = {}
vegfa_gene = next((g for g in Gene.all_genes if g.get_symbol() == "VEGFA"), None)
if vegfa_gene is None:
    raise ValueError("VEGFA not found in Gene.all_genes")
for sample_id in low_inside_df.index:
    meta_row = Meta.get(sample_id)
    tumor_stage = meta_row.get("ajcc_pathologic_tumor_stage", None) if meta_row else None
    cancer = meta_row.get("cancer_type", None) if meta_row else None
    sex = meta_row.get("gender", None) if meta_row else None
    low_tumor_stage_hem[sample_id] = [sample_id, tumor_stage, cancer, sex, vegfa_gene]

for sample_id in low_tumor_stage_hem:
    meta = Meta.get(sample_id) or {}
    stage = meta.get("ajcc_pathologic_tumor_stage")
    tumor_stage = meta.get("ajcc_pathologic_tumor_stage", None) if meta else None
    cancer = meta.get("cancer_type", None) if meta else None
    sex = meta.get("gender", None) if meta else None
    try:
        vegfa_val = vegfa_gene.get_value(sample_id)
    except Exception:
        vegfa_val = None
    low_tumor_stage_hem[sample_id] = [tumor_stage, cancer, sex, vegfa_val]

list(islice(low_tumor_stage_hem, 5))
```

Out[14]:
```
['TCGA-AB-2856-03A-01T-0736-13',
 'TCGA-AB-2812-03A-01T-0734-13',
 'TCGA-AB-2885-03A-01T-0735-13',
 'TCGA-AB-2944-03A-01T-0740-13',
 'TCGA-AB-2986-03A-01T-0734-13']
```

Comparing High and Low VEGFA Expressions Correlated to PIGF Expressions with Cancer Stages

In [15]:
```
import statistics

high = []
low = []

for sample_id in high_tumor_stage_hem:
    tumor_stage = high_tumor_stage_hem[sample_id][0]
    if tumor_stage == '':
        tumor_stage = "0"
    high.append(tumor_stage)

for sample_id in low_tumor_stage_hem:
    tumor_stage = low_tumor_stage_hem[sample_id][0]
    if tumor_stage == '':
```

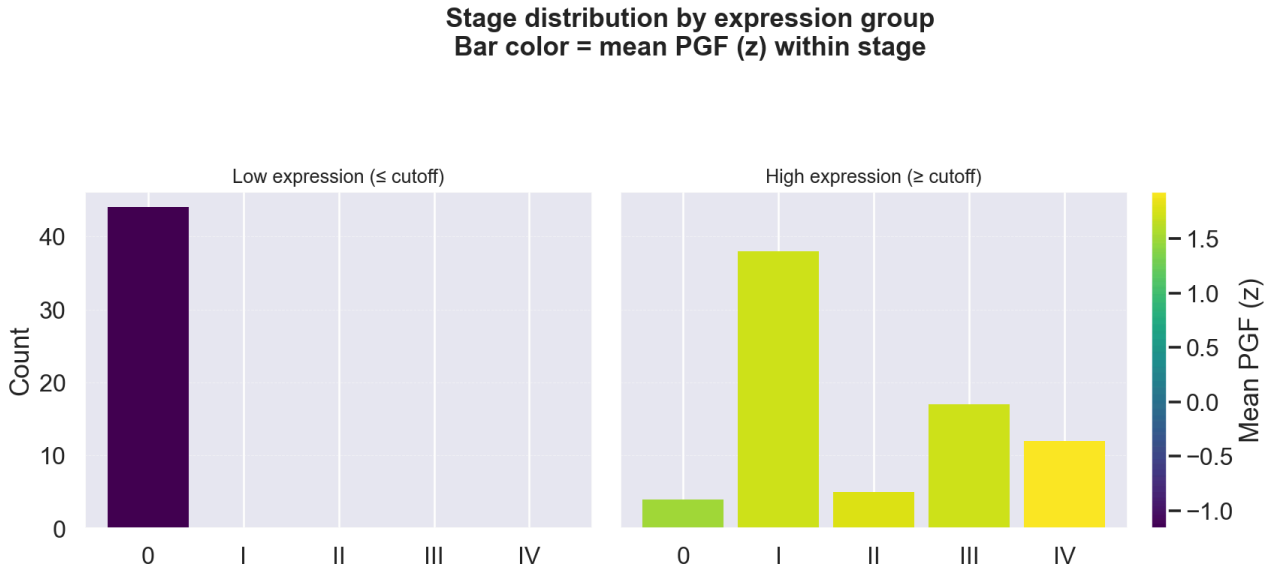
```
        tumor_stage = "0"
    low.append(tumor_stage)
```

In [16]:

```
import re
from collections import defaultdict
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors as mcolors, cm

# Normalize stage strings
_rx = re.compile(r"(?:stage\s*)?(?P<roman>0|i{1,3}|iv)\b\.?\s*[- ]?\s*(?P<letter>[a-d])?",
                 flags=re.IGNORECASE)
def normalize_stage(s):
    if s is None:
        return "0"
    s = str(s).strip()
    if s == "" or s.startswith("[Not "):
        return "0"
    m = _rx.search(s)
    if not m:
        return "0"
    roman  = (m.group("roman") or "").upper()
    letter = (m.group("letter") or "").upper()
    return (roman + letter) if roman else "0"

#  choose PIGF column
pgf_col = "PGF" if "PGF" in Xz.columns else ("PIGF" if "PIGF" in Xz.columns else None)
if pgf_col is None:
    raise ValueError("Neither 'PGF' nor 'PIGF' found in Xz.columns.")

# collect stage & PIGF for inside_df
def collect_stage_pgf(df_inside):
    stage_list, pgf_list = [], []
    for sid in df_inside.index:
        meta = Meta.get(sid) or {}
        stg = normalize_stage(meta.get("ajcc_pathologic_tumor_stage"))
        try:
            val = float(Xz.loc[sid, pgf_col])
        except Exception:
            val = np.nan
        stage_list.append(stg)
        pgf_list.append(val)
    return np.array(stage_list, dtype=object), np.array(pgf_list, dtype=float)

stg_hi, pgf_hi = collect_stage_pgf(high_inside_df)
stg_lo, pgf_lo = collect_stage_pgf(low_inside_df)

# stage order present
stage_order_full = ["0",
    "I","IA","IB","IC",
    "II","IIA","IIB","IIC",
    "III","IIIA","IIIB","IIIC","IIID",
    "IV","IVA","IVB","IVC","IVD"
]
present = [s for s in stage_order_full if (s in set(stg_hi) or s in set(stg_lo))]
if not present:
    raise ValueError("No parseable stages after normalization.")

# counts
def counts_by_stage(stages, present):
    from collections import Counter
    ct = Counter(stages)
    return np.array([ct.get(s, 0) for s in present], dtype=int)

high_counts = counts_by_stage(stg_hi, present)
low_counts  = counts_by_stage(stg_lo, present)

# mean PIGF per stage
def mean_by_stage(stages, values, present):
    means = []
    for s in present:
        mask = (stages == s) & np.isfinite(values)
        means.append(float(np.mean(values[mask])) if mask.any() else np.nan)
    return np.array(means, dtype=float)
```

```
mean_pgf_hi = mean_by_stage(stg_hi, pgf_hi, present)
mean_pgf_lo = mean_by_stage(stg_lo, pgf_lo, present)

# shared color scale across both panels
all_means = np.concatenate([mean_pgf_hi[np.isfinite(mean_pgf_hi)],
                            mean_pgf_lo[np.isfinite(mean_pgf_lo)]], axis=0)
if all_means.size == 0:
    vmin, vmax = -1.0, 1.0
else:
    vmin, vmax = np.percentile(all_means, [2, 98])
    if np.isclose(vmin, vmax):
        vmin, vmax = vmin - 0.5, vmax + 0.5

norm = mcolors.Normalize(vmin=vmin, vmax=vmax)
cmap = cm.get_cmap("viridis")

def colors_from_means(means):
    cols = []
    for m in means:
        cols.append(cmap(norm(m)) if np.isfinite(m) else (0.85,0.85,0.85,1.0))
    return cols

colors_lo = colors_from_means(mean_pgf_lo)
colors_hi = colors_from_means(mean_pgf_hi)

# plot
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(13.5, 5.4), dpi=140, sharey=True)

x = np.arange(len(present))

# Low
ax1.bar(x, low_counts, width=0.85, color=colors_lo, edgecolor="none")
ax1.set_title("Low expression (≤ cutoff)", fontsize=13)
ax1.set_ylabel("Count")
ax1.set_xticks(x); ax1.set_xticklabels(present)
ax1.grid(axis="y", linestyle="--", linewidth=0.5, alpha=0.5)

# High
ax2.bar(x, high_counts, width=0.85, color=colors_hi, edgecolor="none")
ax2.set_title("High expression (≥ cutoff)", fontsize=13)
ax2.set_xticks(x); ax2.set_xticklabels(present)
ax2.grid(axis="y", linestyle="--", linewidth=0.5, alpha=0.5)

# suptitle
fig.suptitle(
    f"Stage distribution by expression group\nBar color = mean {pgf_col} (z) within stage",
    y=1.04, fontsize=18, fontweight="bold"
)

# Colorbar
sm = cm.ScalarMappable(norm=norm, cmap=cmap); sm.set_array([])
# leave right margin for the colorbar
fig.tight_layout(rect=[0, 0, 0.88, 0.95])
cbar = fig.colorbar(sm, ax=[ax1, ax2], location="right", fraction=0.045, pad=0.02)
cbar.set_label(f"Mean {pgf_col} (z)")

plt.show()
```

```
C:\Users\hudso\AppData\Local\Temp\ipykernel_24452\3060712910.py:87: MatplotlibDeprecationWarning: The get_cmap functi
on was deprecated in Matplotlib 3.7 and will be removed in 3.11. Use ``matplotlib.colormaps[name]`` or ``matplotlib.c
olormaps.get_cmap()`` or ``pyplot.get_cmap()`` instead.
  cmap = cm.get_cmap("viridis")
```



Four-Step Pipeline Code Blocks: Test Data

High and Low VEGFA SVM UMAP Boundary on Test Data

In [17]:

```
Gene.instantiate_from_csv("TEST_SET_GSE62944_subsample_log2TPM.csv")
Meta.instantiate_from_csv("TEST_SET_GSE62944_metadata.csv")

'''High Expression UMAP SVM Boundary Analysis'''
# Build expression matrix from Gene/Meta objects
gene_df = pd.DataFrame(
    {sid: {g.get_symbol(): g.get_value(sid) for g in Gene.all_genes}
     for sid in Gene.sample_ids}
).T
gene_df = gene_df.loc[:, ~gene_df.columns.duplicated(keep="first")]

meta_df = pd.DataFrame(Meta.as_rows()).set_index("sample")
common = gene_df.index.intersection(meta_df.index)
gene_df = gene_df.loc[common].copy()
meta_df = meta_df.loc[common].copy()

# Angiogenesis markers
target_genes = ["VEGFA","VEGFB","VEGFC","PGF","ANGPT2","FGF2","PDGFB","IL8","DLL4"]
present = [g for g in target_genes if g in gene_df.columns]
missing = [g for g in target_genes if g not in present]
if missing:
    print("Warning: missing markers not found:", ", ".join(missing))
if not present:
    raise ValueError("None of the requested angiogenesis genes are present.")

# Subset and numeric
X = gene_df[present].apply(pd.to_numeric, errors="coerce")
y = meta_df["cancer_type"].astype(str)

# Scale and UMAP
scaler = StandardScaler()
Xz = pd.DataFrame(scaler.fit_transform(X), index=X.index, columns=X.columns)

umap_model = UMAP(n_neighbors=15, min_dist=0.1, metric="euclidean", random_state=42)
U = umap_model.fit_transform(Xz)

plot_df = pd.DataFrame({"UMAP1": U[:,0], "UMAP2": U[:,1], "Cancer": y.values}, index=X.index)

# ------------------------------ Gating inputs --------------------------------
genes_available = list(Xz.columns)
print("Available genes:", ", ".join(genes_available))
gene = input("Choose a gene to gate (exact symbol from above): ").strip()
if gene not in genes_available:
```

```
    raise ValueError(f"Gene '{gene}' not found in Xz columns.")

mode = input("Threshold mode ('quantile' or 'value'): ").strip().lower()
if mode not in ("quantile", "value"):
    raise ValueError("mode must be 'quantile' or 'value'.")

direction = input("Gate 'high' or 'low' expression? ").strip().lower()
if direction not in ("high", "low"):
    raise ValueError("direction must be 'high' or 'low'.")

expr = Xz.loc[plot_df.index, gene]  # z-scored expression for the chosen gene

if mode == "quantile":
    q = float(input("Quantile in [0,1] (e.g., 0.95 for high or 0.05 for low): ").strip())
    cut = float(expr.quantile(q))
else:
    cut = float(input("Absolute z-score cutoff (e.g., 0.0): ").strip())

# Build labels: 1 = INSIDE (selected), 0 = OUTSIDE
if direction == "high":
    y_bin = (expr >= cut).astype(int).to_numpy()
    thr_text = f"{gene} high ({'q' if mode=='quantile' else 'z'}={q if mode=='quantile' else cut:.2f}, z≥{cut:.2f})"
else:
    y_bin = (expr <= cut).astype(int).to_numpy()
    thr_text = f"{gene} low ({'q' if mode=='quantile' else 'z'}={q if mode=='quantile' else cut:.2f}, z≤{cut:.2f})"

if y_bin.sum() == 0 or y_bin.sum() == len(y_bin):
    raise ValueError("Chosen cutoff produced a single class. Pick a different quantile/value.")

# SVM in UMAP space
C_val_in = input("SVM C (default 1.0): ").strip()
C_val = 1.0 if C_val_in == "" else float(C_val_in)

coords = plot_df[["UMAP1","UMAP2"]].to_numpy()
clf = SVC(kernel="rbf", C=C_val, gamma="scale", probability=False, random_state=42)
clf.fit(coords, y_bin)

# Grid for smooth boundary
pad = 0.08
x_min, x_max = coords[:,0].min(), coords[:,0].max()
y_min, y_max = coords[:,1].min(), coords[:,1].max()
dx, dy = x_max - x_min, y_max - y_min
x_min, x_max = x_min - pad*dx, x_max + pad*dx
y_min, y_max = y_min - pad*dy, y_max + pad*dy

nx = 600
ny = int(nx * (dy/dx if dx > 0 else 1.0))
xx, yy = np.meshgrid(np.linspace(x_min, x_max, nx),
                     np.linspace(y_min, y_max, ny))
grid = np.c_[xx.ravel(), yy.ravel()]
zz = clf.decision_function(grid).reshape(ny, nx)

inside_mask = clf.predict(coords).astype(bool)
inside_df = plot_df.loc[inside_mask]
outside_df = plot_df.loc[~inside_mask]

# Plot side-by-side; legend shows ONLY inside cancers
sns.set_theme(context="talk")
fig, (ax1, ax2) = plt.subplots(
    1, 2, figsize=(16, 7), dpi=140, sharex=True, sharey=True, constrained_layout=True
)

# Consistent colors
labels_all = sorted(plot_df["Cancer"].unique().tolist())
palette = sns.color_palette("tab20", n_colors=max(20, len(labels_all)))
color_map = {lbl: palette[i % len(palette)] for i, lbl in enumerate(labels_all)}

# Left: original UMAP
sns.scatterplot(
    data=plot_df, x="UMAP1", y="UMAP2",
    hue="Cancer", edgecolor="none", s=28, alpha=0.9, ax=ax1,
    legend=False, palette=color_map
)
ax1.set_title("UMAP of Selected Angiogenesis Genes")
ax1.set_xlabel("UMAP1"); ax1.set_ylabel("UMAP2")
```

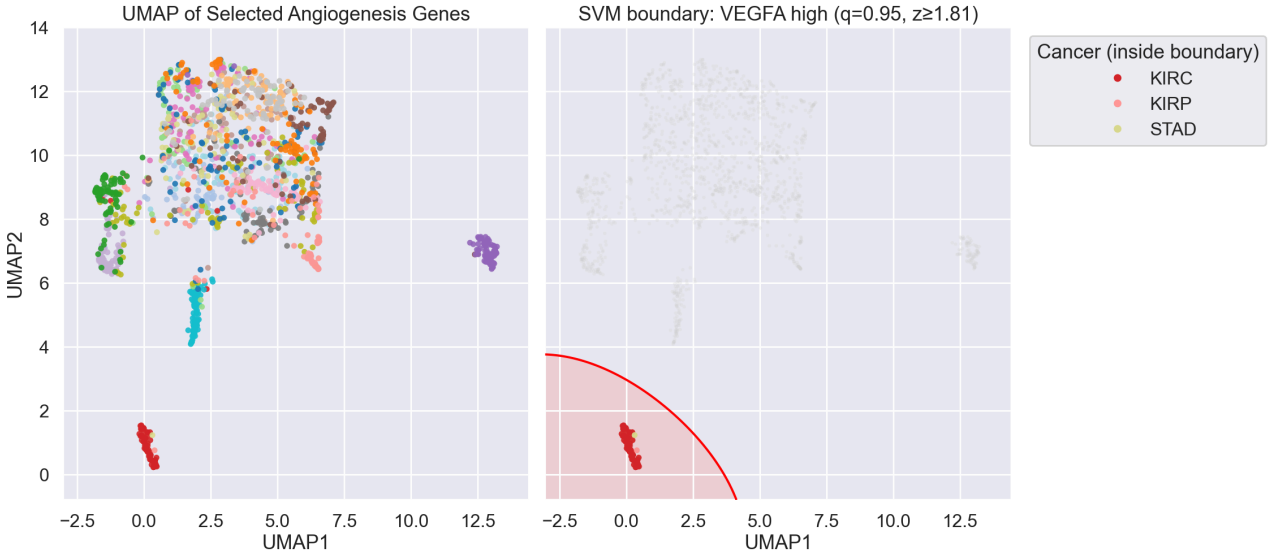
```
# Right: boundary + inside points + legend filtered to inside only
ax2.scatter(plot_df["UMAP1"], plot_df["UMAP2"],
            s=10, c="lightgray", alpha=0.25, edgecolors="none", zorder=1)

# Light fill and red boundary
ax2.contourf(xx, yy, zz, levels=[0, zz.max()], colors=["red"], alpha=0.12, zorder=0)
ax2.contour(xx, yy, zz, levels=[0], colors="red", linewidths=2.0, zorder=3)

sns.scatterplot(
    data=inside_df, x="UMAP1", y="UMAP2",
    hue="Cancer", s=28, edgecolor="none", alpha=0.95, ax=ax2,
    legend=False, palette=color_map
)

# Title shows the selected gene + exact threshold you entered
ax2.set_title(f"SVM boundary: {thr_text}")
ax2.set_xlabel("UMAP1"); ax2.set_ylabel("UMAP2")

# Legend ONLY for cancers inside the boundary
inside_labels = sorted(inside_df["Cancer"].unique().tolist())
handles_right = [Line2D([0],[0], marker='o', linestyle='', markersize=7,
                        markerfacecolor=color_map[lbl], markeredgecolor='none')
                 for lbl in inside_labels]
if inside_labels:
    ax2.legend(
        handles_right, inside_labels,
        title="Cancer (inside boundary)", loc="upper left",
        bbox_to_anchor=(1.02, 1.0), frameon=True, borderaxespad=0.5, ncol=1
    )
else:
    ax2.legend([], [], title="Cancer (inside boundary)")

# Consistent limits
ax1.set_xlim(x_min, x_max)
ax1.set_ylim(y_min, y_max)

plt.show()

# --- Points inside the SVM boundary → DataFrame ------------------------------
coords = plot_df[["UMAP1", "UMAP2"]].to_numpy()
scores = clf.decision_function(coords)
inside_mask = scores > 0

high_inside_df = (
    plot_df.loc[inside_mask]
           .join(Xz.loc[plot_df.index, [gene]], how="left")
           .rename(columns={gene: f"{gene}_z"})
           .copy()
)

high_outside_df = (
    plot_df.loc[~inside_mask]
           .join(Xz.loc[plot_df.index, [gene]], how="left")
           .rename(columns={gene: f"{gene}_z"})
           .copy()
)

print(f"Inside boundary: {high_inside_df.shape[0]} samples")
print(f"Outside boundary: {high_outside_df.shape[0]} samples")



'''Low Expression UMAP SVM Boundary Analysis'''
# Build expression matrix from Gene/Meta objects
gene_df = pd.DataFrame(
    {sid: {g.get_symbol(): g.get_value(sid) for g in Gene.all_genes}
     for sid in Gene.sample_ids}
).T
gene_df = gene_df.loc[:, ~gene_df.columns.duplicated(keep="first")]

meta_df = pd.DataFrame(Meta.as_rows()).set_index("sample")
common = gene_df.index.intersection(meta_df.index)
gene_df = gene_df.loc[common].copy()
meta_df = meta_df.loc[common].copy()
```

```python
# Angiogenesis markers
target_genes = ["VEGFA","VEGFB","VEGFC","PGF","ANGPT2","FGF2","PDGFB","IL8","DLL4"]
present = [g for g in target_genes if g in gene_df.columns]
missing = [g for g in target_genes if g not in present]
if missing:
    print("Warning: missing markers not found:", ", ".join(missing))
if not present:
    raise ValueError("None of the requested angiogenesis genes are present.")

# Subset and numeric
X = gene_df[present].apply(pd.to_numeric, errors="coerce")
y = meta_df["cancer_type"].astype(str)

# Scale and UMAP
scaler = StandardScaler()
Xz = pd.DataFrame(scaler.fit_transform(X), index=X.index, columns=X.columns)

umap_model = UMAP(n_neighbors=15, min_dist=0.1, metric="euclidean", random_state=42)
U = umap_model.fit_transform(Xz)

plot_df = pd.DataFrame({"UMAP1": U[:,0], "UMAP2": U[:,1], "Cancer": y.values}, index=X.index)

# ------------------------------ Gating inputs --------------------------------
genes_available = list(Xz.columns)
print("Available genes:", ", ".join(genes_available))
gene = input("Choose a gene to gate (exact symbol from above): ").strip()
if gene not in genes_available:
    raise ValueError(f"Gene '{gene}' not found in Xz columns.")

mode = input("Threshold mode ('quantile' or 'value'): ").strip().lower()
if mode not in ("quantile", "value"):
    raise ValueError("mode must be 'quantile' or 'value'.")

direction = input("Gate 'high' or 'low' expression? ").strip().lower()
if direction not in ("high", "low"):
    raise ValueError("direction must be 'high' or 'low'.")

expr = Xz.loc[plot_df.index, gene]  # z-scored expression for the chosen gene

if mode == "quantile":
    q = float(input("Quantile in [0,1] (e.g., 0.95 for high or 0.05 for low): ").strip())
    cut = float(expr.quantile(q))
else:
    cut = float(input("Absolute z-score cutoff (e.g., 0.0): ").strip())

# Build labels: 1 = INSIDE (selected), 0 = OUTSIDE
if direction == "high":
    y_bin = (expr >= cut).astype(int).to_numpy()
    thr_text = f"{gene} high ({'q' if mode=='quantile' else 'z'}={q if mode=='quantile' else cut:.2f}, z≥{cut:.2f})"
else:
    y_bin = (expr <= cut).astype(int).to_numpy()
    thr_text = f"{gene} low ({'q' if mode=='quantile' else 'z'}={q if mode=='quantile' else cut:.2f}, z≤{cut:.2f})"

if y_bin.sum() == 0 or y_bin.sum() == len(y_bin):
    raise ValueError("Chosen cutoff produced a single class. Pick a different quantile/value.")

# SVM in UMAP space
C_val_in = input("SVM C (default 1.0): ").strip()
C_val = 1.0 if C_val_in == "" else float(C_val_in)

coords = plot_df[["UMAP1","UMAP2"]].to_numpy()
clf = SVC(kernel="rbf", C=C_val, gamma="scale", probability=False, random_state=42)
clf.fit(coords, y_bin)

# Grid for smooth boundary
pad = 0.08
x_min, x_max = coords[:,0].min(), coords[:,0].max()
y_min, y_max = coords[:,1].min(), coords[:,1].max()
dx, dy = x_max - x_min, y_max - y_min
x_min, x_max = x_min - pad*dx, x_max + pad*dx
y_min, y_max = y_min - pad*dy, y_max + pad*dy

nx = 600
ny = int(nx * (dy/dx if dx > 0 else 1.0))
```

```
xx, yy = np.meshgrid(np.linspace(x_min, x_max, nx),
                     np.linspace(y_min, y_max, ny))
grid = np.c_[xx.ravel(), yy.ravel()]
zz = clf.decision_function(grid).reshape(ny, nx)

inside_mask = clf.predict(coords).astype(bool)
inside_df = plot_df.loc[inside_mask]
outside_df = plot_df.loc[~inside_mask]

# Plot side-by-side; legend shows ONLY inside cancers
sns.set_theme(context="talk")
fig, (ax1, ax2) = plt.subplots(
    1, 2, figsize=(16, 7), dpi=140, sharex=True, sharey=True, constrained_layout=True
)

# Consistent colors
labels_all = sorted(plot_df["Cancer"].unique().tolist())
palette = sns.color_palette("tab20", n_colors=max(20, len(labels_all)))
color_map = {lbl: palette[i % len(palette)] for i, lbl in enumerate(labels_all)}

# Left: original UMAP
sns.scatterplot(
    data=plot_df, x="UMAP1", y="UMAP2",
    hue="Cancer", edgecolor="none", s=28, alpha=0.9, ax=ax1,
    legend=False, palette=color_map
)
ax1.set_title("UMAP of Selected Angiogenesis Genes")
ax1.set_xlabel("UMAP1"); ax1.set_ylabel("UMAP2")

# Right: boundary + inside points + legend filtered to inside only
ax2.scatter(plot_df["UMAP1"], plot_df["UMAP2"],
            s=10, c="lightgray", alpha=0.25, edgecolors="none", zorder=1)

# Light fill and red boundary
ax2.contourf(xx, yy, zz, levels=[0, zz.max()], colors=["red"], alpha=0.12, zorder=0)
ax2.contour(xx, yy, zz, levels=[0], colors="red", linewidths=2.0, zorder=3)

sns.scatterplot(
    data=inside_df, x="UMAP1", y="UMAP2",
    hue="Cancer", s=28, edgecolor="none", alpha=0.95, ax=ax2,
    legend=False, palette=color_map
)

# Title shows the selected gene + exact threshold you entered
ax2.set_title(f"SVM boundary: {thr_text}")
ax2.set_xlabel("UMAP1"); ax2.set_ylabel("UMAP2")

# Legend ONLY for cancers inside the boundary
inside_labels = sorted(inside_df["Cancer"].unique().tolist())
handles_right = [Line2D([0],[0], marker='o', linestyle='', markersize=7,
                        markerfacecolor=color_map[lbl], markeredgecolor='none')
                 for lbl in inside_labels]
if inside_labels:
    ax2.legend(
        handles_right, inside_labels,
        title="Cancer (inside boundary)", loc="upper left",
        bbox_to_anchor=(1.02, 1.0), frameon=True, borderaxespad=0.5, ncol=1
    )
else:
    ax2.legend([], [], title="Cancer (inside boundary)")

# Consistent limits
ax1.set_xlim(x_min, x_max)
ax1.set_ylim(y_min, y_max)

plt.show()

# --- Points inside the SVM boundary → DataFrame ------------------------------
coords = plot_df[["UMAP1", "UMAP2"]].to_numpy()
scores = clf.decision_function(coords)
inside_mask = scores > 0

low_inside_df = (
    plot_df.loc[inside_mask]
           .join(Xz.loc[plot_df.index, [gene]], how="left")
```

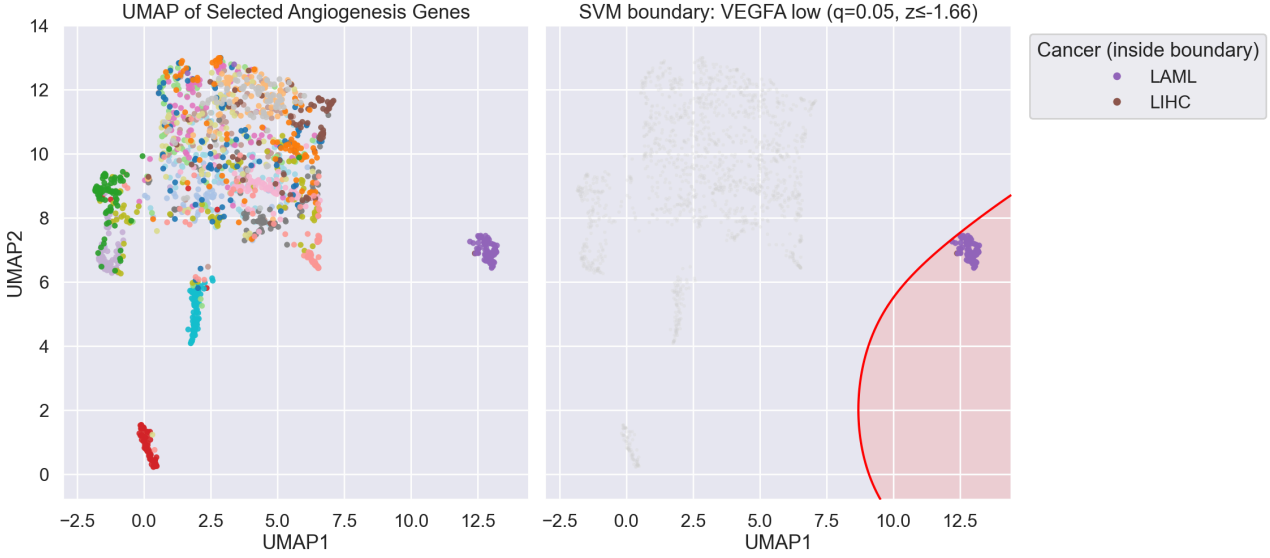
```
            .rename(columns={gene: f"{gene}_z"})
            .copy()
)

low_outside_df = (
    plot_df.loc[~inside_mask]
            .join(Xz.loc[plot_df.index, [gene]], how="left")
            .rename(columns={gene: f"{gene}_z"})
            .copy()
)

print(f"Inside boundary: {low_inside_df.shape[0]} samples")
print(f"Outside boundary: {low_outside_df.shape[0]} samples")
```

```
c:\Users\hudso\anaconda3\Lib\site-packages\umap\umap_.py:1952: UserWarning: n_jobs value 1 overridden to 1 by setting random_state. Use no seed for parallelism.
  warn(
Available genes: VEGFA, VEGFB, VEGFC, PGF, ANGPT2, FGF2, PDGFB, IL8, DLL4
```




```
Inside boundary: 75 samples
Outside boundary: 1525 samples
```

```
c:\Users\hudso\anaconda3\Lib\site-packages\umap\umap_.py:1952: UserWarning: n_jobs value 1 overridden to 1 by setting random_state. Use no seed for parallelism.
  warn(
Available genes: VEGFA, VEGFB, VEGFC, PGF, ANGPT2, FGF2, PDGFB, IL8, DLL4
```




```
Inside boundary: 81 samples
Outside boundary: 1519 samples
```

Creating a Boundary Data Frame

In [20]:
```
from itertools import islice

high_expression_meta = []
for sample_id in high_inside_df.index:
    meta_row = Meta.get(sample_id)
    high_expression_meta.append(meta_row)

high_tumor_stage_hem = {}
vegfa_gene = next((g for g in Gene.all_genes if g.get_symbol() == "VEGFA"), None)
if vegfa_gene is None:
    raise ValueError("VEGFA not found in Gene.all_genes")
for sample_id in high_inside_df.index:
    meta_row = Meta.get(sample_id)
    tumor_stage = meta_row.get("ajcc_pathologic_tumor_stage", None) if meta_row else None
    cancer = meta_row.get("cancer_type", None) if meta_row else None
    sex = meta_row.get("gender", None) if meta_row else None
    high_tumor_stage_hem[sample_id] = [sample_id, tumor_stage, cancer, sex, vegfa_gene]

for sample_id in high_tumor_stage_hem:
    meta = Meta.get(sample_id) or {}
    stage = meta.get("ajcc_pathologic_tumor_stage")
    tumor_stage = meta.get("ajcc_pathologic_tumor_stage", None) if meta else None
    cancer = meta.get("cancer_type", None) if meta else None
    sex = meta.get("gender", None) if meta else None
    try:
        vegfa_val = vegfa_gene.get_value(sample_id)
    except Exception:
        vegfa_val = None
    high_tumor_stage_hem[sample_id] = [tumor_stage, cancer, sex, vegfa_val]

list(islice(high_tumor_stage_hem, 5))
```

Out[20]:
```
['TCGA-A3-3349-01A-01R-1188-07',
 'TCGA-A3-3331-01A-02R-1325-07',
 'TCGA-BP-4326-01A-01R-1289-07',
 'TCGA-B8-A54H-01A-11R-A33J-07',
 'TCGA-B8-4148-01A-02R-1325-07']
```

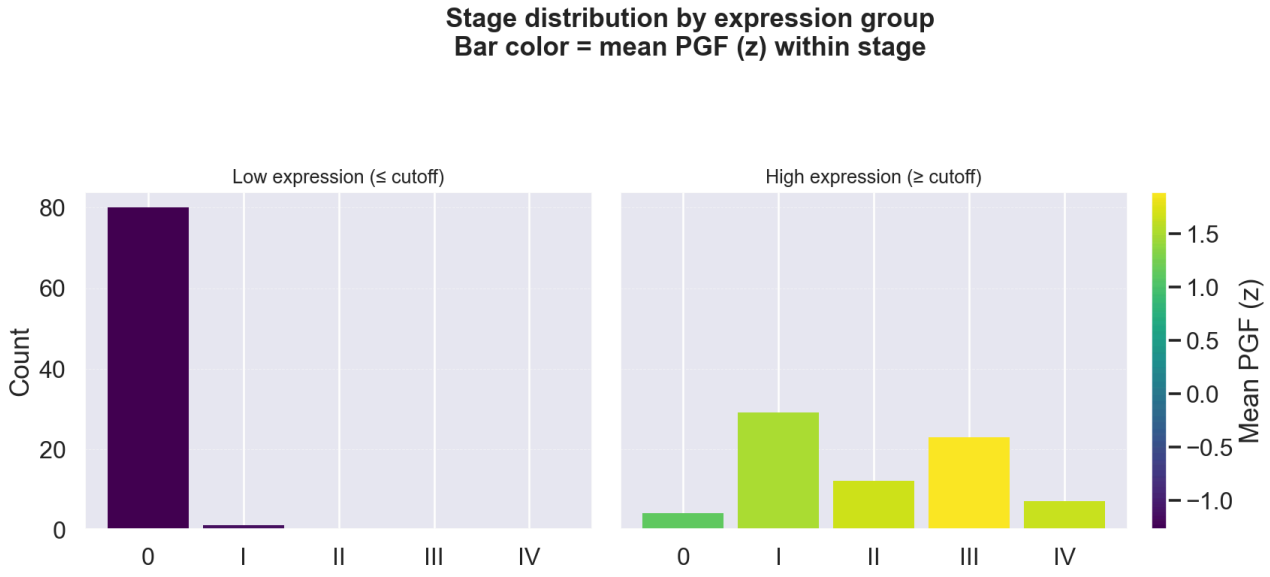
Comparing High and Low VEGFA Expressions Correlated to PIGF Expressions with Cancer Stages

In [21]:
```
import re
from collections import defaultdict
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors as mcolors, cm

# Normalize stage strings
_rx = re.compile(r"(?:stage\s*)?(?P<roman>0|i{1,3}|iv)\b\.?\s*[- ]?\s*(?P<letter>[a-d])?",
                 flags=re.IGNORECASE)
def normalize_stage(s):
    if s is None:
        return "0"
    s = str(s).strip()
    if s == "" or s.startswith("[Not "):
        return "0"
    m = _rx.search(s)
    if not m:
        return "0"
    roman  = (m.group("roman") or "").upper()
    letter = (m.group("letter") or "").upper()
    return (roman + letter) if roman else "0"

#  choose PIGF column
pgf_col = "PGF" if "PGF" in Xz.columns else ("PIGF" if "PIGF" in Xz.columns else None)
if pgf_col is None:
    raise ValueError("Neither 'PGF' nor 'PIGF' found in Xz.columns.")

# collect stage & PIGF for inside_df
def collect_stage_pgf(df_inside):
    stage_list, pgf_list = [], []
    for sid in df_inside.index:
```

```
        meta = Meta.get(sid) or {}
        stg = normalize_stage(meta.get("ajcc_pathologic_tumor_stage"))
        try:
            val = float(Xz.loc[sid, pgf_col])
        except Exception:
            val = np.nan
        stage_list.append(stg)
        pgf_list.append(val)
    return np.array(stage_list, dtype=object), np.array(pgf_list, dtype=float)

stg_hi, pgf_hi = collect_stage_pgf(high_inside_df)
stg_lo, pgf_lo = collect_stage_pgf(low_inside_df)

# stage order present
stage_order_full = ["0",
    "I","IA","IB","IC",
    "II","IIA","IIB","IIC",
    "III","IIIA","IIIB","IIIC","IIID",
    "IV","IVA","IVB","IVC","IVD"
]
present = [s for s in stage_order_full if (s in set(stg_hi) or s in set(stg_lo))]
if not present:
    raise ValueError("No parseable stages after normalization.")

# counts
def counts_by_stage(stages, present):
    from collections import Counter
    ct = Counter(stages)
    return np.array([ct.get(s, 0) for s in present], dtype=int)

high_counts = counts_by_stage(stg_hi, present)
low_counts  = counts_by_stage(stg_lo, present)

# mean PIGF per stage
def mean_by_stage(stages, values, present):
    means = []
    for s in present:
        mask = (stages == s) & np.isfinite(values)
        means.append(float(np.mean(values[mask])) if mask.any() else np.nan)
    return np.array(means, dtype=float)

mean_pgf_hi = mean_by_stage(stg_hi, pgf_hi, present)
mean_pgf_lo = mean_by_stage(stg_lo, pgf_lo, present)

# shared color scale across both panels
all_means = np.concatenate([mean_pgf_hi[np.isfinite(mean_pgf_hi)],
                            mean_pgf_lo[np.isfinite(mean_pgf_lo)]], axis=0)
if all_means.size == 0:
    vmin, vmax = -1.0, 1.0
else:
    vmin, vmax = np.percentile(all_means, [2, 98])
    if np.isclose(vmin, vmax):
        vmin, vmax = vmin - 0.5, vmax + 0.5

norm = mcolors.Normalize(vmin=vmin, vmax=vmax)
cmap = cm.get_cmap("viridis")

def colors_from_means(means):
    cols = []
    for m in means:
        cols.append(cmap(norm(m)) if np.isfinite(m) else (0.85,0.85,0.85,1.0))
    return cols

colors_lo = colors_from_means(mean_pgf_lo)
colors_hi = colors_from_means(mean_pgf_hi)

# plot
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(13.5, 5.4), dpi=140, sharey=True)

x = np.arange(len(present))

# Low
ax1.bar(x, low_counts, width=0.85, color=colors_lo, edgecolor="none")
ax1.set_title("Low expression (≤ cutoff)", fontsize=13)
ax1.set_ylabel("Count")
```

```
ax1.set_xticks(x); ax1.set_xticklabels(present)
ax1.grid(axis="y", linestyle="--", linewidth=0.5, alpha=0.5)

# High
ax2.bar(x, high_counts, width=0.85, color=colors_hi, edgecolor="none")
ax2.set_title("High expression (≥ cutoff)", fontsize=13)
ax2.set_xticks(x); ax2.set_xticklabels(present)
ax2.grid(axis="y", linestyle="--", linewidth=0.5, alpha=0.5)

# suptitle
fig.suptitle(
    f"Stage distribution by expression group\nBar color = mean {pgf_col} (z) within stage",
    y=1.04, fontsize=18, fontweight="bold"
)

# Colorbar
sm = cm.ScalarMappable(norm=norm, cmap=cmap); sm.set_array([])
# leave right margin for the colorbar
fig.tight_layout(rect=[0, 0, 0.88, 0.95])
cbar = fig.colorbar(sm, ax=[ax1, ax2], location="right", fraction=0.045, pad=0.02)
cbar.set_label(f"Mean {pgf_col} (z)")

plt.show()
```

```
C:\Users\hudso\AppData\Local\Temp\ipykernel_24452\3060712910.py:87: MatplotlibDeprecationWarning: The get_cmap functi
on was deprecated in Matplotlib 3.7 and will be removed in 3.11. Use ``matplotlib.colormaps[name]`` or ``matplotlib.c
olormaps.get_cmap()`` or ``pyplot.get_cmap()`` instead.
  cmap = cm.get_cmap("viridis")
```



Verify and validate your analysis:

There are not a lot of options for validating manifold learning as it is an active area of research. We validated our UMAPs by transforming test data to the UMAP model. We found the high and low VEGFA expression samples in our UMAPs. Then we created a histogram for PGF, which amplifies VEGFA, comparing the spread of cancer stages in each boundary.

We first compared the UMAPs. The types of cancer that showed in the upper and lower boundary for pre test and post test slightly varied. There were more types of cancers found in the pre test upper boundary, though both the pre test and post test upper boundaries both showed KIRC and KIRP. The percent of samples inside the upper boundary were approximately the same between the pre and post test UMAPs at 4.22% and 4.69%, respectively. The pre and post test lower boundaries were more similar in cancer types and both showed LAML. There was a greater percentage of samples inside the lower boundary for the post test UMAP at 5.06% as opposed to the pre test percentage at 2.44%.

The pre test and post test data histograms show similar spreads, with most of the low expression samples being in stage 0. In the high expression histograms, both have samples spread in all stages of cancer.

Overall, we see similar trends between our original model and the model with the test data ran through, so we can assume the original model is fitted well. In the future, we may include a supervised machine learning technique so we can get a more quantifiable validation technique.

Conclusions and Ethical Implications:

Conclusions:

From the TCGA RNA-seq data, the Meta and Gene classes ensure the reproducibility of analyzing cross-tumor status via expressed angiogenic markers. In succession with filtering across the 24 present cancer types via high and low VEGFA expression, with a statistically significant alpha level of 0.05, UMAP dimensional reduction paired with an RBF-SVM boundary cleanly drew boundaries around the upper 95% and lower 5% of VEGFA expression, classifying the 24 various types of cancer across their VEGFA expressions. Upon plotting the high and low data points observed inside the SVM boundary with their corresponding pathological tumor stage and colored with their normalized PIGF expression value via a histogram, low VEGFA expression groups concentrated in Stage 0 show consistency with the non-invasive and early lesions determined by the cancer types present within the SVM boundary; while high VEGFA expression groups disperse across all stages, suggesting an increase in VEGFA expression correlates to tumor progression and metastasis potential. Furthermore, the coloration of PIGF expression co-elevates within the same high VEGFA expression clusters within the histogram and UMAP, supporting that co-regulation can lead to tumor progression and metastasis potential. These conclusions are supported by prior findings that VEGFA overexpression is strongly correlated with advanced tumor grade, metastasis, and poor prognosis across multiple solid cancers (Ferrara et al., Nature Reviews Cancer, 2003).

Ethical Implications:

While the VEGFA clustering with our UMAP supports the positive correlation between expression values and pathological tumor stage, the unsupervised SVM boundary fails to establish causality and clinical predictiveness withint the patient's data, generalizing the diagnosis as a whole. This generalization without further validation increases clinical over-interpretation and can cause further harm to patients without validity. In limiting this over-interpretation, distinguishes between solid and hemotological cancers can be established due to their different VEGFA expressions, confining the SVM model's inference to comparable tumor microenvironments thus limiting biased generalizations. In tandem with limiting the SVM model's biases, its important that its parameters and algorithms are transparent when reporting analysis at the clinical stage, increasing the reproducibility of claims and limiting the disconnect between the patients, public, and research staff.

Limitations and Future Work:

Two main limitations affected our model. The first limitation involves the settings we used to create our UMAPs. For numbers of neighbors, we used 15, and for minimum distance, we used 0.1. After creating our model, we tried altering these values to see how they visually affected our UMAPs. Altering the number of neighbors changed where some clumps appeared on the UMAP and changed the spread of the points slightly, but it was difficult to tell what would be optimal. We varied the minimum distance as well and found that decreasing it could make some neighborhoods more defined. However, if the distance was too small, it introduced a greater range of points in the boundaries, which is not ideal as it groups less similar points. We also learned that changing the SVM value could affect the error in the model, with a greater SVM introducing more error. A greater SVM could be useful for future works that want to find very general trends that do not need high accuracy so they can save computation power. We kept our original settings, but acknowledge they were not ideal. Future work could try to refine these inputs.

The second main limitation was due to this method being unsupervised with no widestream way of verifying the model. We used visual inspection to compare the UMAPs before and after transforming test data on them, but this is not reliable. Now that we have data frames for the low and high expression samples, we can have future work focus on our original project goal of seeing if there is a type of cancer that requires greater angiogenesis to sustain life. We could split up these data frames by cancer type. From there, we would use a supervised learning method such as SVMs to predict the cancer stage based on VEGFA expression level. We can validate these models with the scikit score method. Then, we could compare the expression thresholds that the SVM created to classify stages of cancer between the different types of cancer. For example, we could compare what level of expression would be required to reach stage 4 in BRCA versus another cancer such as KRIC.

Overall, the main goals in future work would to have a more verifiable method to identify patterns in VEGFA expression and predict stages of cancer from these expression values.

NOTES FROM YOUR TEAM:

https://docs.google.com/document/d/1sklFRd3o6O24Q-ALGkUGP7E9VSAJIKwF4VreRsw5Azw/edit?usp=sharing

QUESTIONS FOR YOUR TA:

We have none.